ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΔΥΤΙΚΗΣ ΕΛΛΑΔΑΣ
ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ ΚΑΙ ΟΙΚΟΝΟΜΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΜΜΕ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Νέα γενιά της OpenGL: Η γλώσσα σκίασης GLSL

ΑΙΚΑΤΕΡΙΝΗ ΧΑΧΛΙΟΥΤΑΚΗ

ΕΠΟΠΤΕΥΩΝ ΚΑΘΗΓΗΤΗΣ: Δρ. ΑΘΑΝΑΣΙΟΣ ΚΟΥΤΡΑΣ

ΠΥΡΓΟΣ, 2018

ΠΙΣΤΟΠΟΙΗΣΗ

Πιστοποιείται ότι η πτυχιακή εργασία με θέμα:

«**Νέα γενιά της OpenGL: Η γλώσσα σκίασης GLSL**»

του φοιτητή του Τμήματος ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΜΜΕ

ΑΙΚΑΤΕΡΙΝΗ ΧΑΧΛΙΟΥΤΑΚΗ

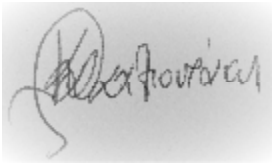
παρουσιάστηκε δημόσια και εξετάσθηκε στο Τμήμα ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΜΜΕ στις

_____ / _____ / _____

Ο ΕΠΙΒΛΕΠΩΝ	Ο ΠΡΟΕΔΡΟΣ ΤΟΥ ΤΜΗΜΑΤΟΣ
Δρ. ΑΘΑΝΑΣΙΟΣ ΚΟΥΤΡΑΣ ΕΠΙΚ. ΚΑΘΗΓΗΤΗΣ	Δρ. ΑΘΑΝΑΣΙΟΣ ΚΟΥΤΡΑΣ ΕΠΙΚ. ΚΑΘΗΓΗΤΗΣ

ΥΠΕΥΘΥΝΗ ΔΗΛΩΣΗ ΠΕΡΙ ΜΗ ΛΟΓΟΚΛΟΠΗΣ

Βεβαιώνω ότι είμαι συγγραφέας αυτής της εργασίας και ότι κάθε βοήθεια την οποία είχα για την προετοιμασία της, είναι πλήρως αναγνωρισμένη και αναφέρεται στην εργασία. Επίσης, έχω αναφέρει τις όποιες πηγές από τις οποίες έκανα χρήση δεδομένων, ιδεών ή λέξεων, είτε αυτές αναφέρονται ακριβώς είτε παραφρασμένες. Ακόμα δηλώνω ότι αυτή η γραπτή εργασία προετοιμάστηκε από εμένα προσωπικά και αποκλειστικά και ειδικά για την συγκεκριμένη πτυχιακή εργασία και ότι θα αναλάβω πλήρως τις συνέπειες εάν η εργασία αυτή αποδειχθεί ότι δεν μου ανήκει.

ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΣΠΟΥΔΑΣΤΗ 1	**ΑΜ**	**ΥΠΟΓΡΑΦΗ**
Αικατερίνη Χαχλιουτάκη	.....30.....	
ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΣΠΟΥΔΑΣΤΗ 2 *(σε περίπτωση που είναι απαραίτητο)*	**ΑΜ**	**ΥΠΟΓΡΑΦΗ**
.....	.....	.....
ΟΝΟΜΑΤΕΠΩΝΥΜΟ ΣΠΟΥΔΑΣΤΗ 3 *(σε περίπτωση που είναι απαραίτητο)*	**ΑΜ**	**ΥΠΟΓΡΑΦΗ**
.....	.....	.....

ΕΥΧΑΡΙΣΤΙΕΣ

Θερμές ευχαριστίες οφείλω στον επιβλέποντα καθηγητή Δρ. Αθανάσιο Κούτρα για τη σημαντική συμβολή του στην ολοκλήρωση της παρούσας εργασίας.

ΠΡΟΛΟΓΟΣ

Τα γραφικά υπολογιστή είναι ένα συναρπαστικό και ταχέως αναπτυσσόμενο πεδίο. Η γοητεία που ασκεί σε όσους ασχοληθούν με αυτό, οφείλεται στη μοναδική μίξη της επιστήμης της μηχανικής με την τέχνη. Στη βιομηχανία των γραφικών υπολογιστή, άνθρωποι με δεξιότητες μηχανικής, σχεδιάζουν λογισμικό γραφικών και προϊόντα hardware που προσφέρουν συνεχώς βελτιούμενα επίπεδα εκτέλεσης και ποιότητας εικόνας. Τα προϊόντα αυτά εμπνέουν ανθρώπους με καλλιτεχνικές δεξιότητες να χρησιμοποιήσουν τα αποτελέσματα που προκύπτουν για να φτιάξουν υπέροχες εικονικές εμπειρίες που ψυχαγωγούν, διδάσκουν, ή βοηθούν άλλους να δημιουργήσουν ή να σχεδιάσουν. Αυτό από την άλλη, εμπνέει τους μηχανικούς να αναπτύξουν ακόμα καλύτερο υλικό(hardware) και λογισμικό(software) προκειμένου να βελτιώσουν τις εμπειρίες της εικόνας που δημιουργήθηκαν από τους καλλιτέχνες. Αυτή η συμβιωτική σχέση μεταξύ μηχανικών και καλλιτεχνών δεν σταμάτησε ποτέ να υπάρχει και έχει φέρει ως αποτέλεσμα, φωτορεαλιστικά εφέ σε ταινίες και εμπειρίες πολύ κοντά στην ποιότητα του σινεμά στα computer games.

Το πιο σημαντικό κομμάτι αυτής της εργασίας, πέρα από το να συστήσει στον αναγνώστη όλο το οικοδόμημα πίσω από την ιστορία των γραφικών, είναι να βοηθήσει στην ανάπτυξη των δεξιοτήτων, όλων όσων επιθυμούν να ασχοληθούν με το κομμάτι της δημιουργίας ποιοτικών εφαρμογών που απαιτούν τη χρήση γραφικών.

[ΠΕΡΙΛΗΨΗ

Η τάση τα τελευταία χρόνια στο υλικό γραφικών είναι η αντικατάσταση της σταθερής γραφικής σωλήνωσης (fixed graphics pipeline) με την προγραμματιζόμενη γραφική σωλήνωση (programmable graphics pipeline) η οποία επιτρέπει τον προγραμματισμό των επιμέρους σταδίων της με αποτέλεσμα την τροποποίηση της λειτουργίας του. Στους σύγχρονους επεξεργαστές γραφικών υπάρχουν τριών ειδών προγραμματιζόμενες μονάδες, οι Επεξεργαστές Κορυφών (Vertex Processors), οι Επεξεργαστές Γεωμετρίας (Geometry Processors) και οι Fragment Processors, κάθε μια από τις οποίες αντικαθιστά ένα κομμάτι της κλασικής σταθερής σωλήνωσης. Κάθε GPU περιέχει δεκάδες ή και εκατοντάδες μονάδες από κάθε κατηγορία για να επιτευχθεί παραλληλισμός στην επεξεργασία. Υπάρχουν GPUs με μονάδες μικρο-επεξεργασίας ροής (streaming processors) γενικού σκοπού που αναλαμβάνουν το ρόλο του vertex ή fragment επεξεργαστή ανάλογα με τις ανάγκες της εφαρμογής. Σήμερα υπάρχουν τρεις γλώσσες υψηλού επιπέδου για προγραμματισμό των μηχανών σκίασης: η Cg (C for graphics), η HLSL (High Level Shading Language) και η GLSL (OpenGL Shading Language). Οι δυο πρώτες αναπτύχθηκαν από κοινού από τις Microsoft και nVidia. Η GLSL, είναι η γλώσσα προγραμματισμού μηχανών σκίασης για το OpenGL API.

Η GLSL έγινε μέρος της OpenGL στην έκδοση 2.0 με την προσθήκη πάνω από 30 συναρτήσεων για το χειρισμό των προγραμμάτων σκίασης. Ο πηγαίος κώδικας του προγράμματος σκίασης στέλνεται από την εφαρμογή στον οδηγό OpenGL (driver). Στη συνέχεια γίνεται η μεταγλώττιση, η σύνδεση και το επακόλουθο φόρτωμα του εκτελέσιμου προγράμματος στους vertex ή fragment επεξεργαστές.

ΣΚΟΠΟΣ ΤΗΣ ΕΡΓΑΣΙΑΣ

Σκοπός της πτυχιακής εργασίας είναι η περιγραφή των προγραμματιστικών μονάδων και των σχετικών αλγορίθμων που υλοποιούνται σε αυτές για τη φωτορεαλιστική φωτοαπόδοση (rendering) συνθετικών τρισδιάστατων σκηνικών σε πραγματικό χρόνο.

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ

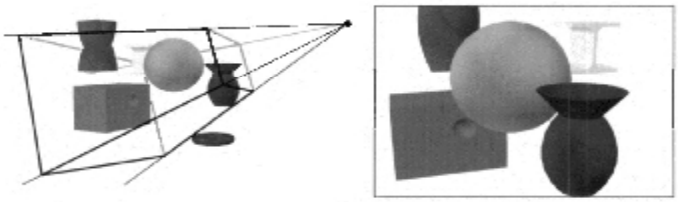
Γραφικά υπολογιστή, OpenGL, GLSL, αλγόριθμοι σκίασης, απεικόνιση υφής, συνθετικές ταινίες

[

ΠΕΡΙΕΧΟΜΕΝΑ

1 ΕΙΣΑΓΩΓΗ ΣΤΗ ΣΩΛΗΝΩΣΗ ΓΡΑΦΙΚΩΝ ΣΤΑΘΕΡΗΣ ΛΕΙΤΟΥΡΓΙΑΣ

(T.Akenine-Möller)Ξεκινώντας σε αυτή την εργασία θα πρέπει πρωτίστως να παρουσιάσουμε αυτό που θεωρείται ότι αποτελεί το βασικό συστατικό των γραφικών σε πραγματικό χρόνο, δηλαδή τον αγωγό απόδοσης γραφικών, γνωστό και ως pipeline. Η κύρια λειτουργία του pipeline είναι να δημιουργήσει ή να αποδόσει μια δισδιάστατη εικόνα, δεδομένης μιας εικονικής κάμερας, τρισδιάστατων αντικειμένων, πηγών φωτισμού, εξισώσεων σκίασης, υφών και πολλών άλλων. Ο αγωγός απόδοσης(pipeline) είναι έτσι το υποκείμενο εργαλείο για την απόδοση σε πραγματικό χρόνο. Η διαδικασία χρήσης του αγωγού απεικονίζεται στην εικόνα 1.1. Οι θέσεις και τα σχήματα των αντικειμένων στην εικόνα καθορίζονται από τη γεωμετρία τους, τα χαρακτηριστικά του περιβάλλοντος και την τοποθέτηση της κάμερας σε αυτό το περιβάλλον. Η εμφάνιση των αντικειμένων επηρεάζεται



από τις ιδιότητες των υλικών, τις πηγές φωτός, τις υφές και τα μοντέλα σκίασης.

Εικόνα 1.1. Στην αριστερή εικόνα, μια εικονική κάμερα βρίσκεται στην άκρη της πυραμίδας (όπου συγκλίνουν τέσσερις γραμμές). Μόνο τα πρωτόκολλα μέσα στον όγκο των προβολών αποδίδονται. Για μια εικόνα που απεικονίζεται σε προοπτική (όπως στην περίπτωση εδώ), ο όγκος της όψης είναι ένας κόλουρος κόνος, δηλ. μια κολοβωμένη πυραμίδα με ορθογώνια βάση. Η δεξιά εικόνα δείχνει τι βλέπει η κάμερα. Σημειώνεται ότι το κόκκινο σχήμα ντόνατς στην αριστερή εικόνα δεν βρίσκεται στην απόδοση στα δεξιά επειδή βρίσκεται έξω από την προβολή του κόλουρου. Επίσης, το μπλε πρίσμα που έχει στρίψει στην αριστερή εικόνα κόβεται στο άνω επίπεδο του κόλουρου.

(Mike Bailey)Στο πρώτο μας μάθημα στα γραφικά του υπολογιστή, χρησιμοποιήσαμε πιθανώς ένα γραφικό API για να μας βοηθήσει να δημιουργήσουμε τα έργα μας. Επειδή αυτή η εργασία επικεντρώνεται σε γραφικά που χρησιμοποιούν το OpenGL, υποθέτουμε ότι το API μας ήταν OpenGL και σε αυτό το κεφάλαιο εξετάζουμε τον αγωγό γραφικών όπως εκφράζεται σε OpenGL εκδόσεις 1.x. Εάν χρησιμοποιήσαμε διαφορετικό API, ειδικά σε ένα πρώτο μάθημα γραφικών, η εμπειρία μας ήταν πιθανώς πολύ κοντά στην προσέγγιση OpenGL. Αυτά τα API χρησιμοποίησαν έναν αγωγό σταθερής λειτουργίας ή έναν αγωγό με σταθερό σύνολο λειτουργιών σε κορυφές και fragments.

1.1 Η ΠΑΡΑΔΟΣΙΑΚΗ ΟΠΤΙΚΗ

Όταν αναπτύσσουμε μια εφαρμογή γραφικών με το API OpenGL, ορίζουμε γεωμετρία, οπτική, προβολή και μια σειρά ιδιοτήτων εμφάνισης. Οι γεωμετρίες των αντικειμένων ορίζονται από τις κορυφές τους, τα normals τους και τα πρωτόκολλα γραφικών τους, που καθορίζονται από ζεύγη `glBegin-glEnd` που περιλαμβάνουν σημεία, γραμμές, συμπιεσμένες ομάδες γεωμετρίας ή πολύγωνα. Κάθε οπτική και προβολή καθορίζονται με μια συγκεκριμένη λειτουργία. Η εμφάνιση καθορίζεται με τον καθορισμό χρώματος, σκίασης, υλικών και φωτισμού ή χαρτογράφησης υφής. Όλες αυτές οι πληροφορίες επεξεργάζονται με πολύ απλό τρόπο από το σταθερό σύστημα OpenGL, που λειτουργεί είτε σε λογισμικό είτε σε κάρτα γραφικών.

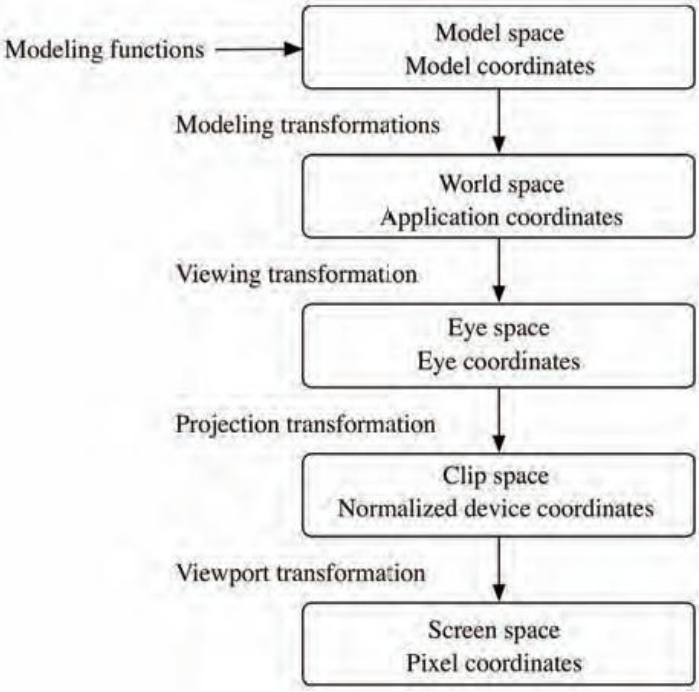
Ο πιο απλός τρόπος για να δούμε τις λειτουργίες του OpenGL είναι να το σκεφτούμε χρησιμοποιώντας δύο συνδεδεμένες λειτουργίες: μια λειτουργία επεξεργασίας κορυφών και μια λειτουργία επεξεργασίας pixel. Κάθε λειτουργία OpenGL σταθερής λειτουργίας έχει ένα προκαθορισμένο σύνολο δυνατοτήτων. Είναι σημαντικό να κατανοήσουμε πώς οι οδηγίες γεωμετρίας και εμφάνισης που δίνουμε εκτελούνται στους αγωγούς. Όταν εργαζόμαστε με shaders, όμως, είναι πολύ σημαντικό να κατανοήσουμε τους αγωγούς. Οι shader μας θα αναλάβουν πραγματικά μέρος αυτών των λειτουργιών, γι' αυτό πρέπει να τους καταλάβουμε.

1.1.1 Η ΛΕΙΤΟΥΡΓΙΑ VERTEX

Για να δημιουργήσουμε τη γεωμετρία μιας σκηνής, καθορίζουμε πρωτόκολλα , κορυφές, και λειτουργίες που δρουν σε κάθε κορυφή και δημιουργούμε τις συντεταγμένες των εικονοστοιχείων στον χώρο της οθόνης. Το πρωτόκολλο που ορίσαμε τότε καθορίζει τα εικονοστοιχεία που πρέπει να συμπληρωθούν για να το αντιπροσωπεύει και κάθε πληροφορία εμφάνισης που ορίσαμε χρησιμοποιείται για να καθορίσει τον τρόπο χρωματισμού αυτών των εικονοστοιχείων στη λειτουργία επεξεργασίας των εικονοστοιχείων. Το τμήμα γεωμετρίας της επεξεργασίας των κορυφών ακολουθεί τη ροή της εικόνας 1.2. Η επεξεργασία γεωμετρίας εκτελείται για κάθε κορυφή ανεξάρτητα από οποιεσδήποτε πληροφορίες σχετικά με την ομαδοποίηση στο καθορισμένο πρωτόκολλό μας. Οι πληροφορίες ομαδοποίησης χρησιμοποιούνται μόνο αφού οι κορυφές ολοκληρώσουν την επεξεργασία των κορυφών.

Το πρώτο στάδιο της λειτουργίας των κορυφών ορίζει τη θεμελιώδη γεωμετρία της σκηνής μας. Η είσοδος σε αυτό το στάδιο είναι το σύνολο των ορισμών κορυφών (κλήσεις λειτουργίας `glVertex *`, `glNormal *` και `glTexCoord *`) και ο ορισμός ομαδοποίησης (κλήσεις λειτουργίας `glBegin (...)` και glEnd ()) που έχουμε ορίσει για τη σκηνή . Κάθε κομμάτι της γεωμετρίας δημιουργείται, ή διαμορφώνεται, στο δικό του χώρο διαμόρφωσης. Αυτός ο χώρος συντεταγμένων μπορεί να είναι οτιδήποτε μας διευκολύνει να ορίσουμε τις κορυφές και τις σχέσεις για το μοντέλο μας. Οι λειτουργίες μοντελοποίησης περιλαμβάνουν οποιεσδήποτε λειτουργίες μπορεί να χρειαστεί για να δημιουργήσουμε αυτές τις διευκρινήσεις και συχνά χρησιμοποιούμε μαθηματικές λειτουργίες που λειτουργούν στο χώρο του μοντέλου. Όπως σημειώσαμε, η γεωμετρία μπορεί να περιλαμβάνει κανονικά διανύσματα και συντεταγμένες υφής, καθώς και συντεταγμένες κορυφών. Περιλαμβάνει επίσης προδιαγραφές πρωτοκόλλων που καθορίζουν τον τρόπο συναρμολόγησης των εικονοστοιχείων από τις κορυφές μας. Μπορεί επίσης να περιλαμβάνει φώτα όταν θέλουμε τα

φώτα να έχουν συγκεκριμένες σχέσεις με τη γεωμετρία μας. Χρησιμοποιούμε πιθανώς άλλες διευκρινήσεις, όπως χρώματα και ιδιότητες υλικών, καθώς ορίζουμε τη γεωμετρία μας. Αυτοί είναι παράγοντες εμφάνισης για τη σκηνή και χρησιμοποιούνται αργότερα στον αγωγό κορυφής, όπως θα δούμε. Η έξοδος αυτού του σταδίου είναι ένα σύνολο κορυφών σε συντεταγμένες μοντέλου, με άλλες γεωμετρικές πληροφορίες και με πληροφορίες πρωτοκόλλου.



Εικόνα 1.2. Επεξεργασία κορυφών στον αγωγό OpenGL.

Το δεύτερο στάδιο της λειτουργίας των κορυφών ορίζει τον παγκόσμιο χώρο που θα κρατήσει ολόκληρη τη σκηνή και θα βάζει όλα τα μεμονωμένα μας μοντέλα σε αυτό το χώρο. Κάθε γεωμετρικό πρωτόκολλο τοποθετείται στον παγκόσμιο χώρο μεταμορφώνοντας μετασχηματισμούς, όπως μετασχηματισμούς κλιμάκωσης, περιστροφής ή μεταφοράς, έτσι ώστε η είσοδος σε αυτό το στάδιο είναι το σύνολο των προδιαγραφών μετασχηματισμού μοντέλων (`glRotatef (...)`, `glTranslatef (...)` και κλήσεις λειτουργίας `glScalef (...)`. Αυτοί οι μετασχηματισμοί μετατρέπουν τις συντεταγμένες ατομικού χώρου μοντέλου σε ένα ενιαίο σύνολο συντεταγμένων χώρου ή κόσμου. Δεν επηρεάζουν τους ορισμούς χρώματος ή υλικού, τις συντεταγμένες υφής ή τις ομαδοποιήσεις, αλλά αλλάζουν τις κορυφές, τα normals και τη γεωμετρία του φωτισμού. Συχνά, τα φώτα ορίζονται απευθείας στον παγκόσμιο χώρο όταν σκεφτόμαστε να φωτίσουμε ολόκληρη τη σκηνή αντί για ένα μόνο αντικείμενο. Η γεωμετρία του φωτός, όπως η θέση ή η κατεύθυνση, επηρεάζεται από οποιαδήποτε μοντελοποίηση ισχύει όταν οριστεί το φως. Η έξοδος αυτού του σταδίου είναι ένα τροποποιημένο σύνολο κορυφών και normals, που αντιπροσωπεύουν την αρχική γεωμετρία σε διαφορετικό χώρο.

Το τρίτο στάδιο της λειτουργίας κορυφής ορίζει τον χώρο των ματιών που δημιουργείται όταν καθορίζουμε πληροφορίες προβολής για τη σκηνή μας. Η είσοδος σε αυτό το στάδιο είναι ο ορισμός μας για το περιβάλλον προβολής, συχνά χρησιμοποιώντας τη λειτουργία GLU `gluLookAt (...)`. Αυτό ορίζει τον μετασχηματισμό προβολής που τροποποιεί τη σκηνή μας για να δημιουργήσει την δεδομένη θέαση του ματιού σε μια σκηνή, ένα σύστημα συντεταγμένων με την οπτική της προέλευσης και τους άξονες x, y και z στους γνωστούς 3D δεξιόχειρους προσανατολισμούς τους. Αυτός ο μετασχηματισμός τροποποιεί την κορυφή, το normal και την πληροφορία φωτισμού, οπότε η έξοδος αυτού του σταδίου είναι η τροποποιημένη γεωμετρία με τις αρχικές πληροφορίες πρωτοκόλλου, με τη γεωμετρία να αντιπροσωπεύει έναν τυπικό χώρο προβολής. Όλες οι πληροφορίες βάθους για μεταγενέστερη επεξεργασία προέρχονται από τις συντεταγμένες z σε αυτό το χώρο του ματιού. Ο πίνακας ***ModelView*** ορίζεται σε αυτό το σημείο, και χρησιμοποιείται για το μετασχηματισμό των κορυφών για γεωμετρικούς υπολογισμούς καθώς και για τη μετατροπή των τιμών των normals, των θέσεων του φωτισμού και των κατευθύνσεων του φωτός για υπολογισμούς φωτισμού.

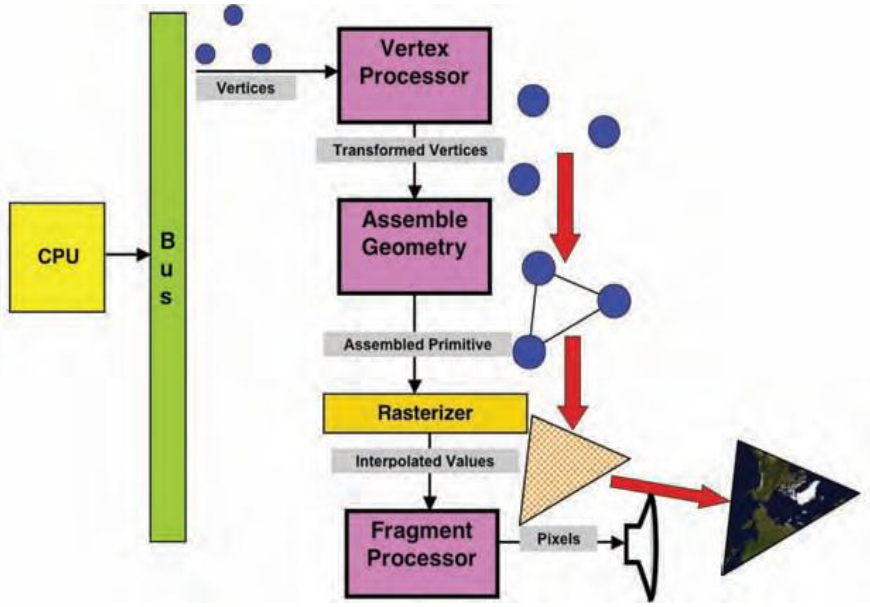
Μόλις βρεθούμε στο χώρο του ματιού, μπαίνουν στο παιχνίδι άλλες πληροφορίες. Ως μέρος του ορισμού κάθε κορυφής, πιθανότατα έχουμε δώσει επίσης κάποιες πληροφορίες εμφάνισης (π.χ. `glColor3f`) ή άλλες πληροφορίες (π.χ. φώτα ή υλικά). Αυτές οι πληροφορίες μπορούν να χρησιμοποιηθούν εδώ για να ορίσουμε το χρώμα κορυφής. Το χρώμα κάθε κορυφής μπορεί να ρυθμιστεί καθώς εφαρμόζονται οι δηλώσεις των χρωμάτων μας ή πραγματοποιούνται οποιεσδήποτε λειτουργίες φωτισμού που καθορίσαμε. Εάν ενεργοποιηθεί ο φωτισμός, οι παράμετροι φωτισμού, η θέση και η κατεύθυνση του φωτός, τα κανονικά διανύσματα και οι προδιαγραφές υλικού χρησιμοποιούνται για τον προσδιορισμό ενός χρώματος για κάθε κορυφή. Κάθε κορυφή θεωρείται ότι έχει μια τιμή χρώματος από το σημείο αυτό στη διαδικασία.

Το τέταρτο στάδιο της λειτουργίας κορυφής ορίζει το χώρο κλιπ που δημιουργείται όταν καθορίζουμε την προβολή της σκηνής μας στο επίπεδο προβολής. Η είσοδος σε αυτό το στάδιο είναι ο ορισμός προβολής μας, είτε προοπτική είτε ορθογραφική. Αυτός ο ορισμός προβολής ορίζει ένα μετασχηματισμό προβολής που πρέπει να εφαρμοστεί στον χώρο του ματιού. Ο ορισμός προβολής δημιουργεί έναν όγκο προβολής και ο μετασχηματισμός προβολής εφαρμόζεται σε αυτόν τον όγκο προβολής για να δημιουργηθεί ένας ορθογώνιος χώρος 3D που μπορεί εύκολα να χρησιμοποιηθεί για το επόμενο στάδιο.

Το τελικό στάδιο της λειτουργίας κορυφής χρησιμοποιεί τις καθορισμένες πληροφορίες παραθύρου προβολής για να δημιουργήσει αναπαραστάσεις χώρου εικονοστοιχείων για κάθε κορυφή στο χώρο της οθόνης. Υπάρχουν δύο κύριες λειτουργίες εδώ. Η μία είναι η αποκοπή(clipping) της γεωμετρίας που καθορίσαμε στα όρια του χώρου κλιπ στον ορισμό προβολής. Αν γίνει αποκοπή, μπορεί να δημιουργηθούν νέα ή τροποποιημένα πρωτόκολλα, καθώς προστίθενται ή διαγράφονται pixels κορυφής. Όταν υπάρχει αποκοπή, τα νέα εικονοστοιχεία κορυφής θα χρειαστεί να έχουν τα νέα τους χρώματα ή να παρεμβάλλονται οι συντεταγμένες υφής με τον ίδιο τρόπο που οι άκρες παρεμβάλλονται στη διαδικασία του rendering. Η δεύτερη λειτουργία είναι η μετατροπή των συντεταγμένων του τρισδιάστατου χώρου κλιπ σε 2D ακέραιες συντεταγμένες του καθορισμένου παραθύρου προβολής. Πρόκειται για μια απλή αναλογική λειτουργία στις συντεταγμένες x και y, συν την ομοιογενή διαίρεση, ακολουθούμενη από μια περικοπή αυτών των πραγματικών τιμών σε ακέραιους αριθμούς. Ταυτοχρόνως, οι z-συντεταγμένες μετατρέπονται σε τιμές βάθους (συνήθως ακέραιοι αριθμοί) που μπορούν να χρησιμοποιηθούν στην απόδοση. Η έξοδος αυτού του σταδίου, και επομένως η έξοδος ολόκληρου του αγωγού κορυφής, είναι ένα σύνολο κορυφών σε ακέραια εικονοστοιχεία x και y με ομαδοποίηση, normals, βάθος,

συντεταγμένες υφής και χρώμα.

Ενώ το Σχήμα 1.2 περιγράφει τις ενέργειες του αγωγού κορυφής, μπορεί επίσης να είναι χρήσιμο να δούμε τις επιδράσεις αυτών των ενεργειών. Στην εικόνα 1.3 περιγράφουμε αυτό δείχνοντας πως ο αγωγός γενικών γραφικών λειτουργεί σε ένα απλό τρίγωνο που αντιπροσωπεύεται από τις τρεις (μπλε) κορυφές. Αυτά αποστέλλονται στον αγωγό κορυφών από την CPU, μετασχηματίζονται σε χώρο οθόνης από τον επεξεργαστή κορυφής, συναρμολογούνται για να μπουν στο ραστεροποιητή και μετατρέπονται σε εικονοστοιχεία από τον επεξεργαστή fragment.



Εικόνα 1.3. Οι δράσεις του συνολικού αγωγού γραφικών.

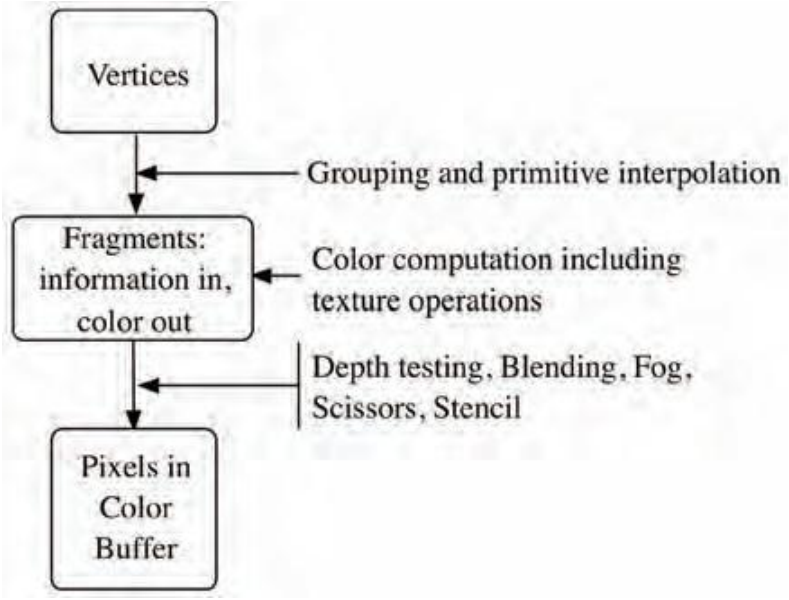
Ο αγωγός γραφικών όπως περιγράφεται παραπάνω περιλαμβάνει έναν αριθμό μετασχηματισμών που σημειώνονται στην εικόνα 1.2: αρκετοί μετασχηματισμοί μοντελοποίησης, ο μετασχηματισμός οπτικής και ο μετασχηματισμός προβολής. Η πραγματική υλοποίηση του αγωγού OpenGL χρησιμοποιεί μια πιο ενιαία έκδοση αυτών των μετασχηματισμών, ωστόσο. Οι μετασχηματισμοί μοντελοποίησης και οπτικής συνδυάζονται στον μετασχηματισμό ModelView και οι νέοι μετασχηματισμοί μοντελοποίησης πολλαπλασιάζονται σε αυτό όπως ορίζονται. Μια στοίβα μετασχηματισμού διατηρείται για τους μετασχηματισμούς ModelView και Projection, με την τρέχουσα έκδοση στην κορυφή της στοίβας. Οι λειτουργίες `glPushMatrix ()` και `glPopMatrix ()` μας επιτρέπουν να αποθηκεύσουμε και να επαναφέρουμε περιβάλλοντα μοντελοποίησης. Ο μετασχηματισμός ModelViewProjection είναι προϊόν των μετασχηματισμών ModelView και Projection και ενημερώνεται κάθε φορά που ενημερώνεται ο μετασχηματισμός ModelView ή ο μετασχηματισμός προβολής. Ο μετασχηματισμός ModelViewProjection εφαρμόζεται σε μεμονωμένες κορυφές για να τις τοποθετήσουμε σε χώρο κλιπ. Το σύστημα διατηρεί επίσης ένα άλλο μετασχηματισμό, τον μετασχηματισμό Normal, που υπολογίζεται ως το αντίστροφο της μεταθέσεως του μετασχηματισμού ModelView, το οποίο χειρίζεται το πρόβλημα της εξασφάλισης ότι το normal μπορεί να χρησιμοποιηθεί σωστά για φωτισμό και άλλες λειτουργίες.

1.1.2 ΤΟ ΤΜΗΜΑ ΕΠΕΞΕΡΓΑΣΙΑΣ ΤΩΝ FRAGMENT ΤΟΥ ΑΓΩΓΟΥ

Πριν από λίγο το ονομάσαμε "επεξεργασία pixel", αλλά το γεγονός είναι ότι πραγματικά ονομάζεται "επεξεργασία fragment". Τι είναι ένα εικονοστοιχείο; Ένα εικονοστοιχείο, στην ορολογία GLSL, είναι ένα σύνολο πληροφοριών εμφάνισης (συνήθως κόκκινο, πράσινο, μπλε, άλφα, βάθος z, κ.λπ.) που πρόκειται να γραφτεί στο framebuffer. Τότε τι είναι ένα fragment; Ένα fragment είναι ένα pixel-to-be. Δηλαδή η τιμή του pixel είναι απαραίτητη για τον υπολογισμό του κόκκινου, του πράσινου, του μπλε, του άλφα, του z-βάθους κλπ. Η λειτουργία ονομάζεται επεξεργασία fragment επειδή η δουλειά του είναι να λάβει όλες αυτές τις πληροφορίες και να παράγει την εμφάνιση του εικονοστοιχείου. Τώρα θα δούμε πώς λειτουργεί αυτή η λειτουργία με ολόκληρο τον αγωγό γραφικών.

Ο αγωγός γραφικών παίρνει τις κορυφές στον χώρο της οθόνης και κατασκευάζει τις περιοχές που ορίσαμε στην ομαδοποίησή μας, με την εμφάνιση που καθορίσαμε στις εντολές rendering του OpenGL. Αυτό περιγράφεται στο ελαφρώς απλοποιημένο διάγραμμα του αγωγού απόδοσης που φαίνεται στην εικόνα 1.4. Αυτό παίρνει ως είσοδο την έξοδο του τελευταίου βήματος του αγωγού κορυφής στην εικόνα 1.2

Εξετάζοντας αυτό, όπως κάναμε στη λειτουργία των κορυφών, ρωτάμε για τις εισόδους και τις εξόδους για κάθε στάδιο. Αρχίζουμε με την έξοδο της λειτουργίας κορυφής: κορυφές σε συντεταγμένες οθόνης με ομαδοποιήσεις, χρώματα, βάθη και συντεταγμένες υφής. (Τα normals δε χρησιμοποιούνται εδώ· ο αγωγός σταθερής λειτουργίας δεν τα χρειάζεται για την επεξεργασία των fragment, επειδή ο φωτισμός υπολογίζεται ανά κορυφή και μόνο οι προκύπτουσες εντάσεις χρώματος παρεμβάλλονται ανά fragment.) Το πρώτο στάδιο rendering παίρνει τις διατεταγμένες κορυφές και δημιουργεί τις ακμές του πρωτοκόλλου. Τα χρώματα, τα βάθη και οι συντεταγμένες υφής στις κορυφές παρεμβάλλονται για να ορίσουν αυτές τις ίδιες ιδιότητες κατά μήκος των άκρων και στη συνέχεια παρεμβάλλονται από αριστερά προς τα δεξιά για κάθε fragment.



Εικόνα 1.4. Μια απλοποιημένη προβολή του αγωγού απόδοσης OpenGL.

Το επόμενο στάδιο rendering επεξεργάζεται fragments. Παίρνει αυτές τις "προ-πληροφορίες" για τις οποίες μόλις συζητήσαμε και δημιουργεί τις πληροφορίες εμφάνισης

που θα γραφτούν στο framebuffer.

Στο τελικό στάδιο του αγωγού γραφικών, το χρώμα του εικονοστοιχείου είναι ενσωματωμένο στο framebuffer με λειτουργίες όπως η δοκιμή βάθους, η ανάμιξη και η κάλυψη που συγκεντρώνουν το τελικό περιεχόμενο framebuffer. Αυτές οι διαδικασίες ενδέχεται να αγνοούν το εικονοστοιχείο (δοκιμή βάθους, κάλυψη) ή να αλλάξουν το χρώμα του εικονοστοιχείου (ανάμειξη). Η τελική έξοδος αυτού του σταδίου είναι το πραγματικό χρώμα στο framebuffer.

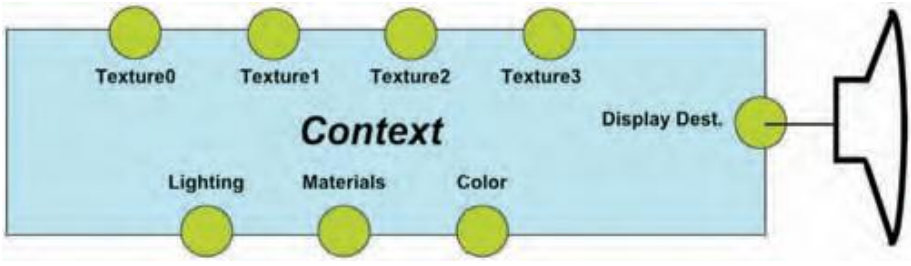
Υπάρχουν, βεβαίως, πολλές λεπτομέρειες σε αυτές τις λειτουργίες και έχουμε σχεδιάσει μόνο τη συνολική διαδικασία εδώ.

1.1.3 ΚΑΤΑΣΤΑΣΗ ΣΤΟΝ ΑΓΩΓΟ ΓΡΑΦΙΚΩΝ

Για τη διαχείριση του μεγάλου αριθμού λειτουργιών του OpenGL και όλων των επιλογών που χρειάζονται, το OpenGL ορίζει και διατηρεί ένα σύνολο πληροφοριών κατάστασης που χρησιμοποιείται στις λειτουργίες κορυφής και rendering. Ένας μεγάλος αριθμός λειτουργιών του OpenGL έχει ως μοναδική λειτουργία τη ρύθμιση των πληροφοριών στην φάση των γραφικών. Καθώς πραγματοποιούνται αυτές οι λειτουργίες, λαμβάνουν τις πληροφορίες τους από αυτή τη φάση.

Πρέπει να γνωρίζουμε πολύ καλά την φάση του OpenGL σε συνεργασία με τους shader, διότι θα πρέπει να αντικαταστήσουμε ορισμένες κρίσιμες λειτουργίες σταθερής λειτουργίας. Θα είναι χρήσιμο να έχουμε μια άνετη γλώσσα και συμβολισμό για να μιλήσουμε για την κατάσταση OpenGL. Εισάγουμε την έννοια ενός πλαισίου γραφικών για να περιγράψουμε την κατάσταση OpenGL και εισάγουμε ένα διάγραμμα αυτού του πλαισίου στην εικόνα 1.5.

Το αρχικό πλαίσιο γραφικών έχει έναν αριθμό προεπιλεγμένων τιμών (π.χ. οι γραμμές είναι λευκές και πλάτους ενός pixel, το χρώμα φόντου είναι μαύρο και δεν υπάρχουν ενεργές υφές.) Όταν ορίζουμε τιμές με λειτουργίες όπως `glColor3f (...)` θα λέμε ότι "αποθέτουμε" την τιμή χρώματος στην υποδοχή που διατηρεί την κύρια τιμή χρώματος στην κατάσταση OpenGL. Αν αλλάξουμε αυτό το χρώμα με μια άλλη κλήση λειτουργίας, τότε η υποδοχή διατηρεί τη νέα τιμή και η παλιά τιμή χάνεται. Έτσι, κάθε "σημείο σύνδεσης"(“docking point”) διατηρεί μια μοναδική τιμή κατάστασης που χρησιμοποιείται στη διαδικασία γραφικών και οι περισσότερες τιμές μπορούν να αναζητηθούν καθώς και να οριστούν.



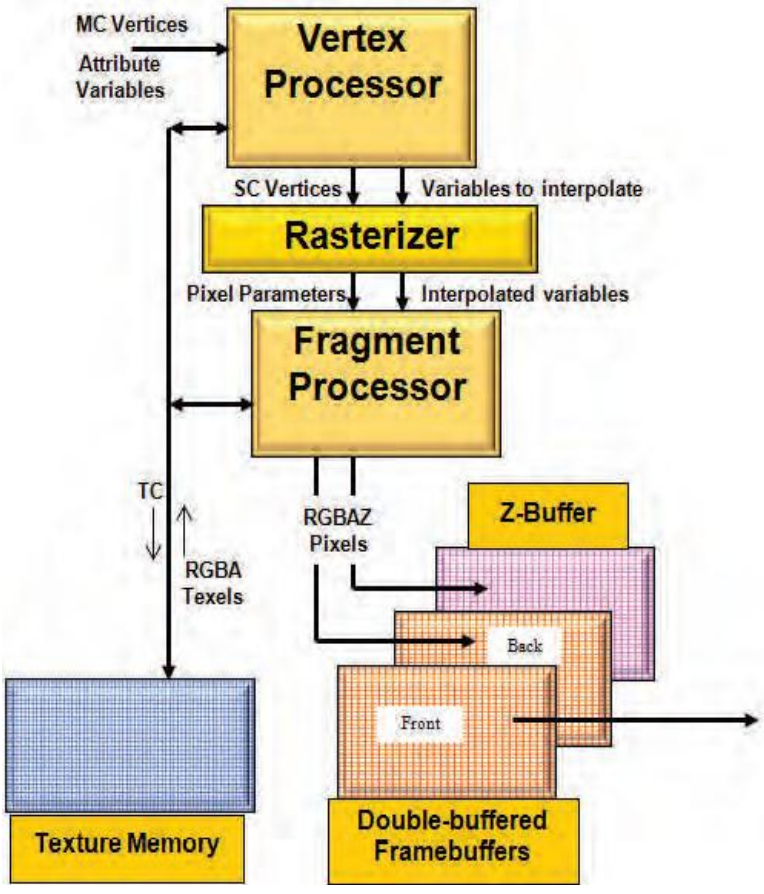
Εικόνα 1.5. Η κατάσταση OpenGL ως αντικείμενο περιβάλλοντος γραφικών.

Στο σύστημα OpenGL, οι πραγματικές διαδικασίες που υλοποιούν τον αγωγό ομαδοποιούνται σε διαφορετικά είδη λειτουργιών. Ένα σχηματικό διάγραμμα αυτών των λειτουργικών ομάδων σε ένα γενικό σύστημα γραφικών παρουσιάζεται στην εικόνα 1.6.

Η πρώτη λειτουργική ομάδα χειρίζεται την επεξεργασία κορυφών που φαίνεται στο σχήμα 1.1. Οι είσοδοι σε αυτή την ομάδα περιλαμβάνουν κορυφές, normals, ορισμούς πρωτοκόλλων, χρώματα, φώτα (και τις παραμέτρους τους), υλικά και συντεταγμένες υφής. Η έξοδος είναι ένα σύνολο κορυφών ως εικονοστοιχεία με το χρώμα, το βάθος και τις συντεταγμένες υφής τους, και ίσως ως αναθεωρημένα πρωτόκολλα.

Το επόμενο βήμα είναι η ραστεροποίηση. Αυτό εφαρμόζει το βήμα Vertices-to-Fragments στον αγωγό rendering της εικόνας 1.4. Η είσοδος στο rasterizer είναι το σύνολο των κορυφών σε συντεταγμένες της οθόνης με τις συντεταγμένες βάθους, χρώματος και υφής, μαζί με τον τρόπο σύνδεσης των κορυφών. Η διαδικασία ραστεροποίησης παρεμβάλλει τις κορυφές για τη δημιουργία fragments και εφαρμόζεται η ίδια παρεμβολή για τον προσδιορισμό των συντεταγμένων βάθους, χρώματος και υφής για κάθε fragment.

Η δεύτερη λειτουργική ομάδα είναι η επεξεργασία fragment. Η είσοδος σε αυτήν την ομάδα είναι ένα ραστεροποιημένο fragment από ένα πρωτόκολλο γραφικών. Το χρώμα του fragment προσδιορίζεται με την επεξεργασία κάθε χρώματος, βάθους και συντεταγμένων της υφής. Η έξοδος της επεξεργασίας fragment είναι ένα σύνολο ολοκληρωμένων εικονοστοιχείων, τα "RGBAZ Pixels" της εικόνας 1.6, με χρώμα (RGB), ανάμειξη (A, για άλφα) και τιμές βάθους (Z) έτοιμα για ενσωμάτωση στο buffer χρώματος .



Εικόνα 1.6. Ο αγωγός OpenGL στο υλικό γραφικών.

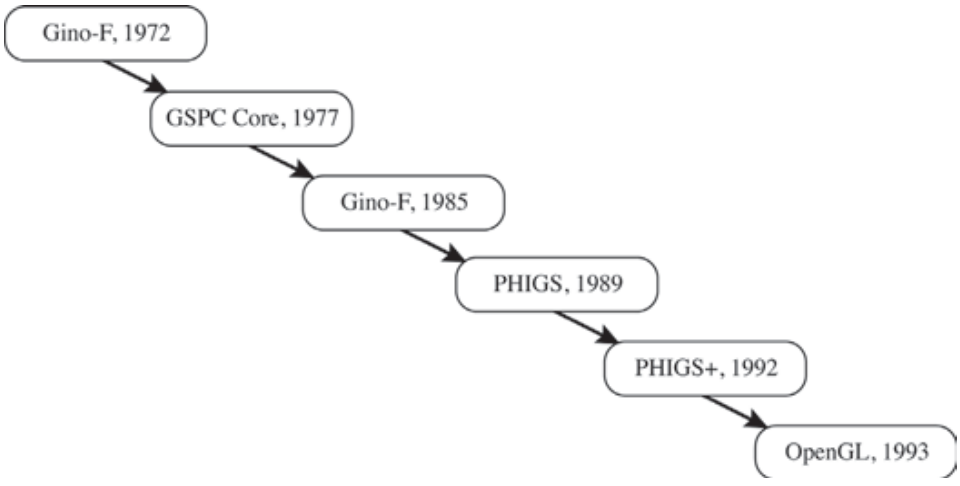
Το τελικό βήμα είναι αυτή η ενσωμάτωση των εικονοστοιχείων στο buffer χρώματος.

Αυτό αντιστοιχεί στην ενότητα Fragments-to-Pixels του αγωγού απόδοσης της εικόνας 1.6. Τα εικονοστοιχεία από τον επεξεργαστή fragment ενσωματώνονται στο buffer χρώματος με λειτουργίες ράστερ που συγχωνεύουν το fragment με τα εικονοστοιχεία στο framebuffer. Είναι το ίδιο για γραφικά με σταθερή λειτουργία και με βάση shader.

1.2 Η ΕΞΕΛΙΞΗ ΤΗΣ ΣΚΙΑΣΗΣ OPEN GL

(Mike Bailey)Στις πρώτες μέρες, τα γραφικά υπολογιστών δεν είχαν πρότυπα μοντέλα προγραμματισμού. Οι προμηθευτές παρείχαν μια διασύνδεση χαμηλού επιπέδου στο υλικό τους και κάθε άτομο ή ομάδα έπειτα ανέπτυξε τη δική του προσέγγιση για τη λήψη πληροφοριών γεωμετρίας και εμφάνισης και την εφαρμογή των συγκεκριμένων αλγορίθμων για τη δημιουργία μιας οθόνης. Ήταν διασκεδαστικό, αλλά δεν ήταν πολύ αποτελεσματικό ή ευέλικτο. Ενώ πολλές από τις εικόνες που δημιουργήθηκαν αυτήν την περίοδο μπορεί να φαίνονται πολύ απλές με βάση τα σημερινά δεδομένα, πολλή δουλειά αφιερώθηκε σε αυτές όμως και οι βασικές ιδέες που δημιουργήθηκαν εκείνες τις μέρες εξακολουθούν να μας επηρεάζουν σήμερα.

Οι πρώτες προσπάθειες για τη μείωση του όγκου των εργασιών ανάπτυξης που απαιτούνται για την παραγωγή επικεντρώθηκαν στα πρότυπα των γραφικών, αλλά τα πρότυπα γενικά παρείχαν μόνο ένα επίπεδο λειτουργιών ελάχιστου κοινού παρονομαστή. Ωστόσο, με την ανάπτυξη προτύπων, οδηγηθήκαμε σε μια όλο και μεγαλύτερη κατανόηση των βασικών λειτουργιών που απαιτούνται στη διαδικασία γραφικών και μας παρασχέθηκε ένα αυξανόμενο επίπεδο προσδοκιών για την ποιότητα των εικόνων που θα μπορούσαν να δημιουργηθούν. Με τη σειρά τους, αυτά οδήγησαν σε μηχανές γραφικών που αναπτύχθηκαν από εταιρείες όπως οι Evans και Sutherland (E & S) και το Silicon Graphics (SGI) και άλλες που άρχισαν να εφαρμόζουν βασικές διαδικασίες γραφικών στο υλικό. Αυτά αύξησαν και πάλι τις προσδοκίες απόδοσης και ποιότητας. Ένα μέρος του "οικογενειακού δένδρου" των μη αποκλειστικών προτύπων γραφικών παρουσιάζεται στην εικόνα 1.7.



Εικόνα 1.7. Ορισμένα πρότυπα γραφικών που οδήγησαν στο OpenGL.

Αρχικά, τα πρότυπα γραφικών προορίζονταν για την επίλυση προβλημάτων φορητότητας. Δηλαδή, τα πρότυπα γραφικών επέτρεψαν στους προγραμματιστές να ανασυνθέσουν υπάρχουσες εφαρμογές σε διαφορετικά συστήματα υλικού με ελάχιστη ποσότητα εργασίας. Όμως, καθώς η επιτάχυνση του υλικού έγινε πιο συνηθισμένη, τα

γραφικά πρότυπα έγιναν επίσης ένα προσχέδιο για τις επιταχύνσεις που χρειάζονταν.

Για παράδειγμα, για να επωφεληθούν οι μηχανές γραφικών SGI, οι μηχανικοί της SGI ανέπτυξαν επίσης ένα API γραφικών που χαρτογραφήθηκε καλά στις διαδικασίες των κινητήρων. Αυτό ήταν το Iris GL, και έκανε τις εφαρμογές γραφικών να αναπτύσσονται πολύ πιο απλά, ώστε να δημιουργηθεί μια έκδοση σε ολόκληρη τη βιομηχανία. Το προκύπτον OpenGL API μπορεί να θεωρηθεί ως ένας από τους βασικούς παράγοντες που έχει κάνει τα γραφικά τόσο διαδεδομένα στον κόσμο της πληροφορικής. Φυσικά, κάποιοι έχουν εξετάσει το OpenGL και έχουν πιστέψει ότι θα μπορούσαν να βελτιώσουν την αντιστοίχιση του API με τις πιο συγκεκριμένες πλατφόρμες τους ή με την επέκταση της λειτουργικότητας του API με διάφορους τρόπους, ώστε να συνεχίσουμε να βρισκόμαστε σε έναν κόσμο με πολλά ανταγωνιστικά " πρότυπα."

Το OpenGL δεν κάνει υποθέσεις στην υποστήριξη υλικού. Είναι δυνατή η πλήρης εφαρμογή του OpenGL στο λογισμικό, χωρίς να επηρεάζονται οι εφαρμογές με οποιονδήποτε τρόπο εκτός από την ταχύτητα. Ωστόσο, πολλές - ίσως οι περισσότερες - εφαρμογές γραφικών πρέπει να δημιουργούν εικόνες με διαδραστικές ταχύτητες. Αυτό ισχύει ιδιαίτερα για εφαρμογές σε πραγματικό χρόνο, όπως παιχνίδια και προσομοίωση. Οι απλές "κάρτες γραφικών" -κάρτες που περιείχαν μνήμη γραφικών και λειτουργούσαν ως απλοί framebuffer- αντικαταστάθηκαν από κάρτες που συμπεριέλαβαν λειτουργίες γραφικών και τελικά τον πλήρη αγωγό γραφικών σταθερής λειτουργίας. Αυτά έδωσαν μεγάλη αύξηση στην ταχύτητα γραφικών, αλλά το κοινό των γραφικών ήθελε περισσότερα.

Ενώ οι απλές κάρτες γραφικών ήταν μια μεγάλη βελτίωση σε σχέση με την απόδοση του λογισμικού, περιορίζονταν σε αυτό που θα μπορούσε να κάνει ο αγωγός σταθερής λειτουργίας και δεν υποστήριζαν πολλά εφέ και δυνατότητες που ένας δημιουργικός προγραμματιστής γραφικών μπορεί να θέλει. Το επόμενο βήμα, όπου είμαστε τώρα, ήταν να κάνουμε τις κάρτες προγραμματιζόμενες έτσι ώστε να μπορούν να προστεθούν επιπλέον λειτουργίες ανάλογα με τις ανάγκες. Με νεοεμφανιζόμενα συστήματα όπως το OpenGL ES (για ενσωματωμένα συστήματα όπως PDA και κινητά τηλέφωνα) που δεν έχουν αγωγό σταθερής λειτουργίας και με το Core OpenGL 4.0 αντικαθιστώντας την προσέγγιση σταθερής λειτουργίας με μια προσέγγιση που απαιτεί shader, φαίνεται σαφές ότι οι shaders είναι ολοένα και πιο κεντρικά στις εφαρμογές γραφικών υπολογιστών και ότι όποιος σχεδιάζει να κάνει σοβαρή εργασία γραφικών θα χρειαστεί να ειδικευτεί στον προγραμματισμό και την ανάπτυξη του Shader.

1.2.1 ΙΣΤΟΡΙΑ ΤΩΝ SHADERS

Παρόλο που οι shaders με βάση τη GPU είναι ένα σχετικά πρόσφατο φαινόμενο, η συνολική ιστορία των shaders πηγαίνει περίπου 30 χρόνια πίσω. Κοιτάζοντας στο παρελθόν, θα μπορούσε να θεωρηθεί ότι ξεκίνησε το 1977, με την εμφάνιση μιας ταινίας χαμηλού προϋπολογισμού που έμελλε να εξελιχθεί σε ένα λατρευτικό φαινόμενο: Star Wars Episode IV: A New Hope.

Το Star Wars IV ήταν επαναστατικό στη χρήση μοντέλων και ρομποτικά ελεγχόμενων καμερών για να δημιουργήσει την ψευδαίσθηση των πραγματικών κινούμενων διαστημικών πλοίων σε μια άγρια μάχη. Χρησιμοποίησε γραφικά υπολογιστή, αλλά όχι πολλά. Αυτό που χρησιμοποίησε ήταν πολύ κάτω από τις δυνατότητες εκείνης της εποχής, αλλά η εκπληκτική επιτυχία του box office της ταινίας απέδειξε ότι ειδικά εφέ πωλούν εισιτήρια. Αλλά για τις μελλοντικές ταινίες, κατέστη σαφές ότι θα ήταν δύσκολο να κλιμακωθεί σε μεγάλο βαθμό η χρήση φυσικών μοντέλων. Ωστόσο, ο George Loukas ήταν ένας άνθρωπος με ένα όραμα - και το πιο σημαντικό, η ταινία του είχε δώσει τα χρήματα για την υλοποίηση αυτού του

οράματος.

Όσον αφορά τα γραφικά των υπολογιστών, ο Lucas προσέλαβε τον Ed Catmull και άλλους από το Ινστιτούτο Τεχνολογίας της Νέας Υόρκης γύρω στο 1980 για να γίνει η Διεύθυνση Υπολογιστών της Lucasfilm. Οι προσπάθειές τους στη Lucasfilm είχαν τρεις άξονες:

• Ψηφιακή επεξεργασία και σύνθεση.

• Υλικό για επεξεργασία εικόνας 2D.

• Υλικό για απόδοση σε 3D γραφικά.

Το 1983, η Lucasfilm έστειλε τις 2D και 3D ομάδες στην εταιρεία τους, που ονομάζεται Pixar, και την πώλησε στον Steve Jobs το 1986. Η ομάδα 2D Image Processing παρήγαγε τον Pixar Image Computer (PIC), μια συσκευή υλικού για την επεξεργασία εικόνας. Το PIC χρησιμοποίησε λειτουργίες SIMD τεσσάρων κατευθύνσεων (πολλαπλών δεδομένων εντολής με μία μόνο εντολή) για ταυτόχρονη επεξεργασία εικόνας και στα τέσσερα στοιχεία RGBA. Έτσι, όταν λέμε στο GLSL

```
vec4 rgba;
. . .
rgba *= 0.5;
```

χρησιμοποιούμε την σύγχρονη εξέλιξη του παραδείγματος PIC SIMD. Παρά την τεχνική επιτυχία της, η Pixar σταμάτησε τελικά τις εργασίες της PIC για να επικεντρωθεί στην απόδοση 3D.

Η πρόθεση της ομάδας rendering της Pixar ήταν να δημιουργήσει μια συσκευή απόδοσης υλικού. Αλλά πρώτα, έπρεπε να αναπτυχθεί ένα πρωτότυπο λογισμικού αυτής της συσκευής. Αυτό ήταν γνωστό ως το σύστημα REYES, ένα αφιέρωμα στο Point Reyes στη βόρεια Καλιφόρνια, αλλά και ένα ακρωνύμιο για τα Renders Everything You Ever Saw.

Εικόνα 1.8. Ο Ιππότης της Βιτρίνας από το “Young Sherlock Holmes”.

Επιγραμματικά να αναφέρουμε ότι το σύστημα REYES βασίστηκε σε μια αρχιτεκτονική, η οποία σχεδιάστηκε με διάφορους στόχους:

- Πολυπλοκότητα / ποικιλομορφία μοντέλου: Για να δημιουργηθούν οπτικά πολύπλοκες και πλούσιες εικόνες, οι χρήστες ενός συστήματος απόδοσης πρέπει να είναι ελεύθεροι να μοντελοποιήσουν μεγάλους αριθμούς (100.000s) σύνθετων γεωμετρικών δομών που ενδεχομένως δημιουργούνται με τη χρήση διαδικαστικών μοντέλων, όπως fractals και συστήματα σωματιδίων.

- Πολυπλοκότητα σκίασης: Ένα μεγάλο μέρος της οπτικής πολυπλοκότητας σε μια σκηνή δημιουργείται από τον τρόπο με τον οποίο οι ακτίνες φωτός αλληλεπιδρούν με επιφάνειες στερεών αντικειμένων. Γενικά, στα γραφικά του υπολογιστή, αυτό διαμορφώνεται χρησιμοποιώντας υφές. Οι υφές μπορούν να είναι έγχρωμες συστοιχίες εικονοστοιχείων, να περιγράφουν μετατοπίσεις επιφάνειας ή διαφάνεια ή ανακλαστικότητα επιφάνειας. Ο Reyes επιτρέπει στους χρήστες να ενσωματώνουν διαδικαστικούς shaders, με τους οποίους επιτυγχάνεται η δομή των επιφανειών και η οπτική αλληλεπίδραση χρησιμοποιώντας προγράμματα υπολογιστή που εφαρμόζουν διαδικαστικούς αλγόριθμους και όχι απλούς πίνακες αναζήτησης. Ένα καλό μέρος του αλγορίθμου στοχεύει στην ελαχιστοποίηση του χρόνου που αφιερώνουν οι επεξεργαστές να ανακτούν τις υφές από τα αποθηκευμένα δεδομένα.

- Ελάχιστο Ray Tracing: Την εποχή που προτάθηκε ο Reyes, τα συστήματα υπολογιστών ήταν σημαντικά λιγότερο ικανά από την άποψη της επεξεργασίας ισχύος και αποθήκευσης. Αυτό σήμαινε ότι το Ray Tracing μιας φωτορεαλιστική σκηνής θα πάρει δεκάδες ή εκατοντάδες ώρες ανά πλαίσιο. Αλγόριθμοι όπως ο Reyes, οι οποίοι γενικά δεν κάνουν Ray Tracing, τρέχουν πολύ πιο γρήγορα με σχεδόν φωτορεαλιστικά αποτελέσματα.

- Ταχύτητα: Το rendering μιας ταινίας διάρκειας δύο ωρών σε 24 καρέ ανά

δευτερόλεπτο σε ένα χρόνο επιτρέπει σε 3 λεπτά rendering ανά καρέ, κατά μέσο όρο.

- Ποιότητα εικόνας: Οποιαδήποτε εικόνα περιέχει ανεπιθύμητα, σχετιζόμενα με το αλγόριθμο αντικείμενα θεωρείται απαράδεκτη.

- Ευελιξία: Η αρχιτεκτονική θα πρέπει να είναι αρκετά ευέλικτη ώστε να ενσωματώνει νέες τεχνικές καθώς καθίστανται διαθέσιμες, χωρίς να απαιτείται η πλήρης επανεισαγωγή του αλγορίθμου.

Το 1984, ο Rob Cook από την Pixar δημοσίευσε το ορόσημό του "Shade Trees" , στο οποίο έδειξε πώς η διαδικασία του rendering θα μπορούσε να διαχειρίζεται από το χρήστη γράφοντας "scripts" και εισάγοντάς τα στα σωστά σημεία του αγωγού rendering.

Μια γενική προσέγγιση για τα Shade Trees είναι η εξάλειψη της εξίσωσης σταθερής σκίασης και ολόκληρης της προσέγγισης δύο σταδίων. Προκειμένου να προσπαθήσουμε να περιγράψουμε όλες τις πιθανές επιφάνειες με μια μόνο εξίσωση, ο shader ενορχηστρώνει ένα σύνολο βασικών λειτουργιών, όπως τα προϊόντα dot και η κανονικοποίηση του διανύσματος. Ο shader οργανώνει αυτές τις πράξεις σε ένα δέντρο. Κάθε λειτουργία είναι ένας κόμβος του δέντρου. Κάθε κόμβος παράγει μία ή περισσότερες παραμέτρους εμφάνισης ως έξοδο και μπορεί να χρησιμοποιήσει μηδενικές ή περισσότερες παραμέτρους εμφάνισης ως είσοδο. Για παράδειγμα, οι εισρεόμενες παράμετροι σε έναν "διάχυτο" κόμβο είναι ένα normal επιφάνειας και ένα διάνυσμα φωτός, και η έξοδος είναι μια έντονη τιμή. Το normal μπορεί να προέρχεται από το γεωμετρικό normal, έναν bump map ή μια διαδικαστική υφή. Η έξοδος μπορεί να είναι η είσοδος σε έναν "πολλαπλασιαστικό κόμβο", ο οποίος θα πολλαπλασίαζε την ένταση με την άλλη είσοδο, ένα χρώμα. Ο shader εκτελεί τους υπολογισμούς στους κόμβους που διασχίζουν το δέντρο σε postorder. Η έξοδος της ρίζας του δέντρου είναι το τελικό χρώμα. Βασικές γεωμετρικές πληροφορίες, όπως η κανονική επιφάνεια και η θέση του αντικειμένου, είναι τα φύλλα του δέντρου. (Γενικά, οι κόμβοι στην πραγματικότητα σχηματίζουν ένα κατευθυνόμενο ακυκλικό γράφημα, επειδή μία μόνο εμφάνιση παραμέτρου μπορεί να χρησιμοποιηθεί ως είσοδος σε περισσότερους από έναν κόμβους.)

Ακόμη και μια παράμετρος εμφάνισης που συνήθως θεωρείται η τελική σκιά μπορεί να αντιμετωπιστεί η ίδια ως ενδιάμεσο βήμα. Αυτό είναι ιδιαίτερα χρήσιμο για την απόδοση μιας επιφάνειας που αποτελείται από διαφορετικά υλικά. Η τελική σκιά μπορεί να είναι ένας συνδυασμός των αποχρώσεων των διαφόρων υλικών, με το ποσό των διαφόρων υλικών που βασίζονται ίσως σε έναν χάρτη υφής.

Τα Shade Trees μπορούν να περιγράψουν ένα ευρύ φάσμα καταστάσεων σκίασης από το πιο απλό έως το πιο περίπλοκο. Διαφορετικοί τύποι των υπολογισμών σκίασης μπορούν να συνυπάρχουν σε μια ενιαία εικόνα, σε κάθε επιφάνεια χρησιμοποιώντας τόσο πολλές ή λίγες λειτουργίες όπως απαιτούνται.

Η ιδέα του Shade Tree επέτρεψε στους προγραμματιστές να δημιουργούν πολλά διαφορετικά εφέ χωρίς να χρειάζεται να προσθέτουν συνεχώς νέο κώδικα μόνιμα στον επεξεργαστή. Μια από τις πρώτες εμπορικές χρήσεις αυτών των shaders ήταν στην ταινία Young Sherlock Holmes το 1985, η οποία δημιούργησε τον ιππότη της βιτρίνας που φαίνεται στην εικόνα 1.8.

Εν τω μεταξύ, συνεχίστηκε η δουλειά σχετικά με την απόδοση υλικού μαζί με το πρωτότυπο λογισμικό REYES. Κάποιος έκανε το σχόλιο ότι κάποια μέρα όλοι θα έχουν μαζί τους ένα μικρό κουτί αποδόσεων στη ζώνη τους. Θα είναι σαν ένα Sony Walkman, είπε, αλλά αντίθετα θα ονομάζεται RenderMan, και έτσι γεννήθηκε το όνομα. Τελικά, η ιδέα του υλικού

μειώθηκε υπέρ μιας λύσης λογισμικού γενικού σκοπού, η οποία έγινε το πακέτο Photorealistic RenderMan (prman).

Ωστόσο, άλλοι πήραν την ιδέα των shaders και ανέπτυξαν διαφορετικές προσεγγίσεις λογισμικού και υλικού για τη δημιουργία γραφικών σκηνών. Το 1985, ο Perlin [34] δημοσίευσε το ορόσημό του Paper Synthesizer. Η χρήση του από μια διαδικαστική λειτουργία θορύβου για να γίνουν οι επιφάνειες πιο ενδιαφέρουσες ίσως κατάφερε περισσότερο να προωθήσει τη χρήση των shaders από οποιαδήποτε άλλη εξέλιξη. Βέβαια, συχνά παραβλέπεται ότι αυτό το έργο δημιούργησε λειτουργίες σκίασης επιφάνειας με εκφράσεις και έλεγχο ροής και έτσι έδειξε στην κοινότητα των γραφικών πόσα θα μπορούσαν να γίνουν με τις γλωσσικές διαδικασίες στον αγωγό γραφικών.

Το 1998, οι Olano και Lastra ανέπτυξαν μια γλώσσα σκίασης για το σύστημα γραφικών PixelFlow. Το PixelFlow ήταν μια πολύ καινοτόμος προσέγγιση των γρήγορων γραφικών που αναπτύχθηκαν στο Πανεπιστήμιο της Βόρειας Καρολίνας. Μερικές από τις ιδέες τους σχετικά με τον παραλληλισμό μπορεί να φανεί ότι επηρέασαν το υλικό γραφικών του σήμερα. Η γλώσσα σκίασης τους έφτασε τα ποσοστά update 30 καρέ / δευτερόλεπτο - πρώτη φορά για μια γλώσσα σκίασης. Το 2001, οι Proudfoot et al. [στο Στάνφορντ ανέπτυξε μια γλώσσα σκίασης υψηλότερου επιπέδου που θα μπορούσε να διαδώσει με διαφάνεια τις λειτουργίες της σε ένα συνδυασμό CPU και GPU, οπουδήποτε είχε νόημα. Αυτό ήταν σημαντικό επειδή επέτρεψε σε έναν προγραμματιστή γραφικών να οδηγήσει την καμπύλη δυνατοτήτων επιτάχυνσης υλικού χωρίς να αλλάξει κώδικα.

Μέχρι τις αρχές της δεκαετίας του 2000, το υλικό γραφικών είχε γίνει πολύ εξελιγμένο και αρκετά γρήγορο ώστε οι άνθρωποι άρχισαν να σκέφτονται ότι χρειάζονται το ίδιο είδος ευέλικτης δυνατότητας σκίασης που ο Rob Cook είχε περιγράψει σχεδόν 20 χρόνια πριν. Οι πρώτες εφαρμογές του ήταν η Cg και η HLSL (Υψηλού Επιπέδου Γλώσσα Shader) , τα οποία, ενώ ήταν ξεχωριστά προϊόντα, αναπτύχθηκαν σε lockstep και έτσι φαίνονται πολύ παρόμοια. Η Cg αναπτύχθηκε από την NVIDIA Corporation, ενώ η HLSL αναπτύχθηκε από τη Microsoft ως μέρος του API γραφικών Direct3D. Σύντομα μετά από αυτό ήρθε το GLSL (OpenGL Shading Language), το οποίο δημιουργήθηκε από το OpenGL Architecture Review Board (ARB).

Αυτές οι τρεις γλώσσες σκίασης προσανατολισμένες στο υλικό κάνουν τα πράγματα λίγο διαφορετικά, αλλά όλα έχουν την ίδια βασική λειτουργικότητα: σκίαση κορυφής, γεωμετρίας και fragment (ή pixel), μια γλώσσα τύπου C και πρόσβαση στις βασικές τιμές δεδομένων μέσα στον αγωγό γραφικών.

1.3 ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ

(Lengyel)Σε όλη την αρχιτεκτονική μηχανών τρισδιάστατων γραφικών, είναι συχνά απαραίτητο να μετασχηματίσουμε ένα σύνολο διανυσμάτων από ένα χώρο συντεταγμένων σε άλλο. Για παράδειγμα, οι συντεταγμένες κορυφών για ένα μοντέλο μπορούν να αποθηκευτούν στο χώρο του αντικειμένου, αλλά πρέπει να μετασχηματιστούν στο χώρο της κάμερας πριν από την απόδοση(rendering) του μοντέλου. Σε αυτό το σημείο, ασχολούμαστε με γραμμικούς μετασχηματισμούς μεταξύ διαφορετικών καρτεσιανών πλαισίων συντεταγμένων. Αυτοί οι μετασχηματισμοί περιλαμβάνουν απλές κλίμακες και μεταφορές, καθώς και αυθαίρετες περιστροφές.

1.3.1 ΓΡΑΜΜΙΚΟΙ ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ

Υποθέτουμε ότι έχουμε δημιουργήσει ένα σύστημα τρισδιάστατων συντεταγμένων *C* που αποτελείται από ένα άξονα προέλευσης(origin) και τρεις συντεταγμένες, όπου ένα σημείο **P** έχει τις συντεταγμένες <x, y, z>. Οι τιμές x, y και z μπορούν να θεωρηθούν ως οι αποστάσεις που πρέπει κανείς να ταξιδέψει κατά μήκος κάθε ενός από τους άξονες συντεταγμένων από την προέλευση για να φτάσει στο σημείο **P**. Υποθέτουμε τώρα ότι εισάγουμε ένα δεύτερο σύστημα συντεταγμένων *C'* στο οποίο συντεταγμένες < x', y', z '> μπορούν να εκφραστούν ως γραμμικές συναρτήσεις των συντεταγμένων <x, y, z> στο C. Δηλαδή, ας υποθέσουμε ότι μπορούμε να γράψουμε

$$x'(x,y,z)= U_1x + V_1y + W_1z + T_1$$
$$y'(x,y,z)= U_2x + V_2y + W_2z + T_2$$
$$z'(x,y,z)= U_3x + V_3y + W_3z + T_3$$

(1.3.1)

Αυτό συνιστά γραμμικό μετασχηματισμό από *C* σε *C'* και μπορεί να γραφεί σε μορφή πίνακα ως εξής

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} U_1 V_1 W_1 \\ U_2 V_2 W_2 \\ U_3 V_3 W_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} + \begin{bmatrix} T_1 \\ T_2 \\ T_3 \end{bmatrix}$$

(1.3.2)

Οι συντεταγμένες *x ', y'* και *z'* μπορούν να θεωρηθούν ως οι αποστάσεις που πρέπει κανείς να ταξιδέψει κατά μήκος των αξόνων στο *C'* για να φτάσει στο σημείο **P**. Το διάνυσμα **T** αντιπροσωπεύει τη μεταφορά από την προέλευση του C στην προέλευση του C', και ο πίνακας του οποίου οι στήλες είναι τα διανύσματα **U**, **V** και **W** αντιπροσωπεύει τον τρόπο αλλαγής του προσανατολισμού των αξόνων συντεταγμένων όταν μετασχηματίζεται από *C* σε *C'*. Υποθέτοντας ότι ο μετασχηματισμός είναι αντιστρέψιμος, ο γραμμικός μετασχηματισμός από *C'* σε *C* δίνεται από

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} U_1 V_1 W_1 \\ U_2 V_2 W_2 \\ U_3 V_3 W_3 \end{bmatrix}^{-1} \left(\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} - \begin{bmatrix} T1 \\ T2 \\ T3 \end{bmatrix} \right)$$

(1.3.3)

Σε αυτό το κομμάτι θα επικεντρωθούμε αποκλειστικά σε γραμμικούς μετασχηματισμούς για τους οποίους **T** $\equiv$ **0**, οπότε τα διανύσματα **U**, **V** και **W** αντιπροσωπεύουν τις εικόνες στο C' των βασικών διανυσμάτων <1,0,0>, <0 , 1, 0> και <0,0,1> στο C.

Οι πολλαπλοί γραμμικοί μετασχηματισμοί μπορούν να συνενωθούν και να αντιπροσωπεύονται από ένα μοναδικό πίνακα και μεταφορά. Για παράδειγμα, οι συντεταγμένες των κορυφών μπορεί να χρειαστεί να μετασχηματιστούν από το χώρο του αντικειμένου και στη συνέχεια από τον παγκόσμιο χώρο στο χώρο της κάμερας. Οι δύο μετασχηματισμοί συνδυάζονται σε ένα ενιαίο μετασχηματισμό που αντιστοιχεί συντεταγμένες χώρου αντικειμένου απευθείας σε συντεταγμένες χώρου κάμερας.

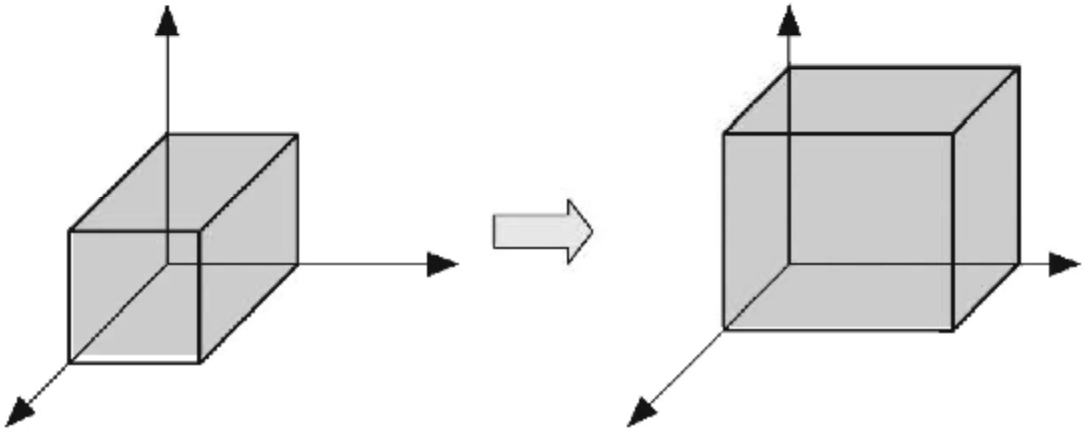
1.3.2 ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ ΚΛΙΜΑΚΑΣ

Για την κλίμακα ενός διανύσματος **P** με συντελεστή *α*, υπολογίζουμε απλώς **P'**= *α* **P**. Σε τρεις διαστάσεις, αυτή η λειτουργία μπορεί επίσης να εκφραστεί ως προϊόν πίνακα

$$P' = \begin{bmatrix} a00 \\ 0a0 \\ 00a \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_Z \end{bmatrix}$$

(1.3.4)

Αυτό ονομάζεται ομοιόμορφη κλίμακα. Αν θέλουμε να κλιμακώσουμε ένα διάνυσμα με διαφορετικές ποσότητες κατά μήκος των αξόνων x, y και z, όπως φαίνεται στην εικόνα 1.9, τότε μπορούμε να χρησιμοποιήσουμε έναν πίνακα παρόμοιο με τον πίνακα ομοιόμορφης κλιμάκωσης αλλά του οποίου οι διαγώνιες καταχωρήσεις δεν είναι αναγκαστικά όλες ίσες. Αυτό ονομάζεται μη ομοιόμορφη κλίμακα και μπορεί να εκφραστεί ως προϊόν πίνακα



Εικόνα 1.9 Ανομοιόμορφη κλιμάκωση

$$P' - \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_Z \end{bmatrix}$$

(1.3.5)

Μια ελαφρώς πιο πολύπλοκη λειτουργία κλιμάκωσης που κάποιος μπορεί να επιθυμεί να εκτελέσει είναι μια μη ομοιόμορφη κλίμακα που εφαρμόζεται σε τρεις αυθαίρετους άξονες. Ας υποθέσουμε ότι θέλουμε να κλιμακώσουμε με συντελεστή *a* κατά μήκος του άξονα **U**, με συντελεστή *b* κατά μήκος του άξονα **V** και με συντελεστή *c* κατά μήκος του άξονα **W**. Τότε μπορούμε να μετασχηματίσουμε από το σύστημα συντεταγμένων (**U, V, W**) στο σύστημα συντεταγμένων (**i, j, k**), εφαρμόζουμε τη λειτουργία κλιμάκωσης σε αυτό το

σύστημα χρησιμοποιώντας την εξίσωση (1.3.5) και, στη συνέχεια, μετασχηματίζουμε πάλι στο σύστημα συντεταγμένων (**U, V, W**). Αυτό μας δίνει το ακόλουθο προϊόν πίνακα.

$$P' = \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix}^{-1} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

(1.3.6)

1.3.3 ΜΕΤΑΣΧΗΜΑΤΙΣΜΟΙ ΠΕΡΙΣΤΡΟΦΗΣ

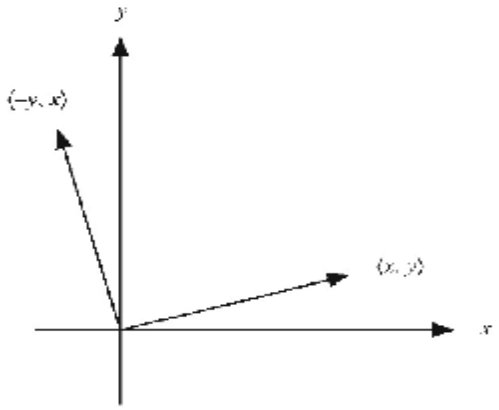
Μπορούμε να βρούμε πίνακες 3 x 3 που περιστρέφουν ένα σύστημα συντεταγμένων σε μια γωνία θ γύρω από τον άξονα x-, y-, ή z- χωρίς πολύ μεγάλη δυσκολία. Θεωρούμε ότι μια περιστροφή με θετική γωνία γύρω από τον άξονα **A** είναι αυτή που εκτελεί περιστροφή αριστερόστροφα όταν ο άξονας **A** δείχνει προς το μέρος μας.

Καταρχάς, θα βρούμε έναν γενικό τύπο για περιστροφές σε δύο διαστάσεις. Όπως φαίνεται στην εικόνα 1.10, μπορούμε να εκτελέσουμε μια περιστροφή κατά 90 μοίρες προς τα αριστερά του 2D διανύσματος **P** στο επίπεδο x - y με την αντικατάσταση των συντεταγμένων x και y και βάζοντας αρνητικό πρόσημο στη νέα συντεταγμένη x. Καλώντας το περιστρεφόμενο διάνυσμα **Q**, έχουμε **Q** = (- P_y , P_x). Τα διανύσματα **P** και **Q** σχηματίζουν μια ορθογώνια βάση για το επίπεδο x-y. Μπορούμε επομένως να εκφράσουμε οποιοδήποτε διάνυσμα στο επίπεδο x-y ως γραμμικό συνδυασμό αυτών των δύο διανυσμάτων. Ειδικότερα, όπως φαίνεται στην εικόνα 1.11, κάθε 2D διάνυσμα **P'** που προκύπτει από την περιστροφή του διανύσματος **P** δια μίας γωνίας θ μπορεί να εκφραστεί με όρους των περιεχομένων του που είναι παράλληλα προς **P** και **Q**. Η βασική τριγωνομετρία μας επιτρέπει να γράψουμε

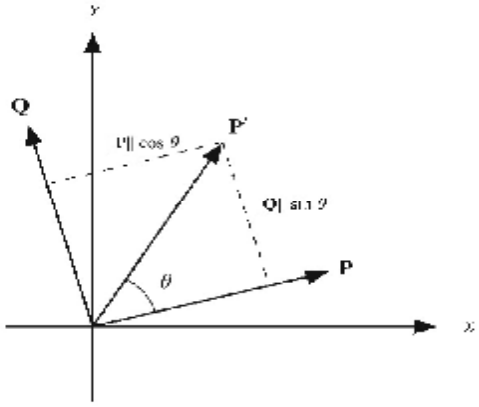
$$\mathbf{P}' = \mathbf{P}\cos\theta + \mathbf{Q}\sin\theta.$$

(1.3.7)

Αυτό μας δίνει τις ακόλουθες εκφράσεις για τα συστατικά του **P** '.



Εικόνα 1.10 Περιστροφή κατά 90 μοίρες στο επίπεδο x-y



Εικόνα 1.11. Ένα περιστρεφόμενο διάνυσμα μπορεί να εκφραστεί ως ο γραμμικός συνδυασμός του αρχικού διανύσματος και της περιστροφής του αρχικού διανύσματος κατά 90 μοίρες προς τα αριστερά.

$$P'_x = P_x \cos\theta - P_y \sin\theta$$

$$P'_y = P_y \cos\theta + P_x \sin\theta$$

(1.3.8)

Μπορούμε να το ξαναγράψουμε σε μορφή πίνακα ως εξής

$$P' = \begin{bmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{bmatrix} P$$

(1.3.9)

Ο πίνακας περιστροφής 2D στην εξίσωση (1.3.9) μπορεί να επεκταθεί σε περιστροφή γύρω από τον άξονα z σε τρεις διαστάσεις, λαμβάνοντας την τρίτη σειρά και στήλη από τον πίνακα ταυτότητας. Αυτό εξασφαλίζει ότι η συντεταγμένη z ενός διανύσματος παραμένει σταθερή κατά τη διάρκεια μιας περιστροφής γύρω από τον άξονα z, όπως θα περιμέναμε. Ο πίνακας $\mathbf{R}_z$ (θ) ο οποίος εκτελεί μια περιστροφή κατά την γωνία θ γύρω από τον άξονα z δίδεται έτσι από

$$R_z(\theta) = \begin{bmatrix} cos\theta & -sin\theta & 0 \\ sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(1.3.10)

Ομοίως, μπορούμε να αντλήσουμε τους ακόλουθους πίνακες 3 x 3 $\mathbf{R}_x$ (θ) και $\mathbf{R}_y$ (θ) που εκτελούν περιστροφές κατά μια γωνία θ γύρω από τους άξονες x και y αντίστοιχα.

$$R_z(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} cos\theta & 0 & sin\theta \\ 0 & 1 & 0 \\ -sin\theta & 0 & cos\theta \end{bmatrix}$$

(1.3.11)

1.3.4 ΠΕΡΙΣΤΡΟΦΗ ΓΥΡΩ ΑΠΟ ΕΝΑΝ ΑΥΘΑΙΡΕΤΟ ΑΞΟΝΑ

Ας υποθέσουμε ότι επιθυμούμε να περιστρέψουμε ένα διάνυσμα **P** διαμέσου μιας γωνίας θ γύρω από έναν αυθαίρετο άξονα του οποίου η κατεύθυνση αντιπροσωπεύεται από ένα μοναδιαίο διάνυσμα **A**. Μπορούμε να αποσυνθέσουμε το διάνυσμα **P** σε συστατικά που είναι παράλληλα προς το **A** και κάθετα στο **A** . Καθώς η παράλληλη συνιστώσα (η προβολή του **P** στο **A**) παραμένει αμετάβλητη κατά τη διάρκεια της περιστροφής, μπορούμε να μειώσουμε το πρόβλημα σε εκείνο της περιστροφής του κάθετου στοιχείου του **P** γύρω από το **A**.

Δεδομένου ότι το **A** είναι ένα μοναδιαίο διάνυσμα, έχουμε τον ακόλουθο απλοποιημένο τύπο για την προβολή του **P** επί το **A**.

$$\text{proj}_{\mathbf{A}}\mathbf{P}=(\mathbf{A}\cdot\mathbf{P})\mathbf{A}$$

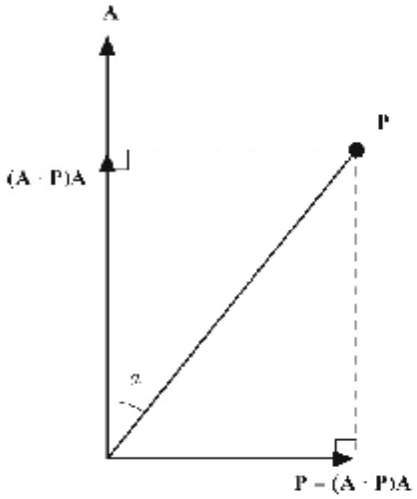
(1.3.12)

Η συνιστώσα του P που είναι κάθετη στο A δίνεται στη συνέχεια από

$$\text{perp}_{\mathbf{A}}\mathbf{P}=\mathbf{P}-(\mathbf{A}\cdot\mathbf{P})\mathbf{A}$$

(1.3.13)

Αφού περιστρέψουμε αυτό το κάθετο στοιχείο γύρω από το **A**, θα προσθέσουμε το σταθερό παράλληλο στοιχείο που δίνεται από την εξίσωση (1.3.12) για να φτάσουμε στην



τελική μας απάντηση.

Εικόνα 1.12 Περιστροφή γύρω από αυθαίρετο άξονα

Η περιστροφή του κάθετου στοιχείου λαμβάνει χώρα στο επίπεδο κάθετα προς τον άξονα **A**. Όπως προηγουμένως, εκφράζουμε το περιστρεφόμενο διάνυσμα ως γραμμική συνένωση του $\text{perp}_{\mathbf{A}}\mathbf{P}$ και του διανύσματος που προκύπτει από περιστροφή του $\text{perp}_{\mathbf{A}}\mathbf{P}$ κατά 90 μοίρες γύρω από το A. Ευτυχώς, μια τέτοια έκφραση είναι εύκολο να βρεθεί. Αφήνουμε το α να είναι η γωνία μεταξύ του αρχικού διανύσματος P και του άξονα A. Να σημειωθεί ότι το μήκος του $\text{perp}_{\mathbf{A}}\mathbf{P}$ είναι ίσο με $\|\mathbf{P}\|\sin\alpha$ επειδή σχηματίζει την πλευρά απέναντι από τη γωνία α που φαίνεται στην εικόνα 1.12. Ένα διάνυσμα του ίδιου μήκους που δείχνει προς την

κατεύθυνση που θέλουμε δίνεται από το **A** x **P**.

Μπορούμε τώρα να εκφράσουμε την περιστροφή του $\text{perp}_{\mathbf{A}}\mathbf{P}$ μέσω μιας γωνίας θ ως

$$[\mathbf{P} - (\mathbf{A} \cdot \mathbf{P})\,\mathbf{A}]\cos\theta + (\mathbf{A} \times \mathbf{P})\sin\theta.$$

(1.3.14)

Η προσθήκη $\text{proj}_{\mathbf{A}}\mathbf{P}$ σε αυτό μας δίνει την ακόλουθη έκφραση για την περιστροφή του αρχικού διανύσματος **P** γύρω από τον άξονα **A**.

$$\mathbf{P}' = \mathbf{P}\cos\theta + (\mathbf{A} \times \mathbf{P})\sin\theta + \mathbf{A}(\mathbf{A} \cdot \mathbf{P})(1 - \cos\theta)$$

(1.3.15)

Αντικαθιστώντας τα A x P και A (A · P) στην εξίσωση (1.3.15) με τα αντίστοιχα του πίνακα που δίδονται από τις εξισώσεις (1.3.16) και (1.3.17)

$$proj_Q P = \frac{1}{\|Q\|^2}\begin{bmatrix} Q_x^2 Q_x & Q_y Q_x & Q_z \\ Q_x Q_y & Q_y^2 Q_y & Q_z \\ Q_x Q_z & Q_y Q_z & Q_z^2 \end{bmatrix}\begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}$$

(1.3.16)

$$P \times Q = \begin{bmatrix} 0 & -P_z & P_y \\ P_z & 0 & -P_x \\ -P_y & P_x & 0 \end{bmatrix}\begin{bmatrix} Q_x \\ Q_y \\ Q_z \end{bmatrix}$$

(1.3.17)

Αντίστοιχα, έχουμε

$$P' = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} P cos\theta + \begin{bmatrix} 0 & -A_z & A_y \\ A_z & 0 & -A_x \\ -A_y & A_x & 0 \end{bmatrix} P sin\theta + \begin{bmatrix} A_x^2 A_x & A_y A_x & A_z \\ A_x A_y & A_y^2 A_y & A_z \\ A_x A_z & A_y A_z & A_z^2 \end{bmatrix} P\,(1 - cos\theta)$$

(1.3.18)

Ο συνδυασμός αυτών των όρων και ο καθορισμός $c = cos\theta$ και $s = sin\theta$ μας δίνει τον ακόλουθο τύπο για τον πίνακα $\mathbf{R}_A(\theta)$ που περιστρέφει ένα διάνυσμα από μια γωνία θ γύρω από τον άξονα **A**.

$$R_A(\theta) = \begin{bmatrix} c + (1-c)A_x^2 & (1-c)A_xA_y & -sA_z(1-c)A_xA_z + sA_y \\ (1-c)A_x & A_y + sA_zc + (1-c) & A_y^2(1-c)A_yA_z - sA_x \\ (1-c)A_xA_z & -sA_y(1-c) & A_yA_z + sA_xc + (1-c)A_z^2 \end{bmatrix}$$

(1.3.19)

1.3.5 ΟΜΟΓΕΝΕΙΣ ΣΥΝΤΕΤΑΓΜΕΝΕΣ

Μέχρι τώρα, έχουμε ασχοληθεί μόνο με μετασχηματισμούς που μπορούν να εκφραστούν ως η λειτουργία ενός πίνακα 3 x 3 σε ένα διάνυσμα τριών διαστάσεων. Μια σειρά τέτοιων μετασχηματισμών θα μπορούσε να αναπαρασταθεί από έναν απλό πίνακα 3 x 3 ίσο με το προϊόν των πινάκων που αντιστοιχούν στους μεμονωμένους μετασχηματισμούς. Ένας σημαντικός μετασχηματισμός που έχει αφεθεί έξω είναι η λειτουργία μεταφοράς. Ένα σύστημα συντεταγμένων μεταφέρεται στο χώρο χωρίς να επηρεάζεται αλλιώς ο προσανατολισμός ή η κλίμακα των αξόνων με την απλή προσθήκη ενός μετατοπισμένου διανύσματος. Αυτή η λειτουργία δεν μπορεί να εκφραστεί με τη μορφή πίνακα 3 x 3. Έτσι, για να μετατρέψουμε ένα σημείο **P** από το ένα σύστημα συντεταγμένων στο άλλο, συνήθως βρισκόμαστε στην εκτέλεση της λειτουργίας

$$\mathbf{P'} = \mathbf{MP} + \mathbf{T},$$

(1.3.20)

όπου M είναι κάποιος μετατρέψιμος πίνακας 3 x 3 και T είναι ένα διάνυσμα μεταφοράς 3D. Διεξάγοντας δύο πράξεις του τύπου που παρουσιάζεται στην εξίσωση (1.3.20), προκύπτει η μάλλον μπερδεμένη εξίσωση

$$\begin{aligned} \mathbf{P'} &= \mathbf{M}_2(\mathbf{M}_1\mathbf{P} + \mathbf{T}_1) + \mathbf{T}_2 \\ &= (\mathbf{M}_2\mathbf{M}_1)\mathbf{P} + \mathbf{M}_2\mathbf{T}_1 + \mathbf{T}_2 \end{aligned}$$

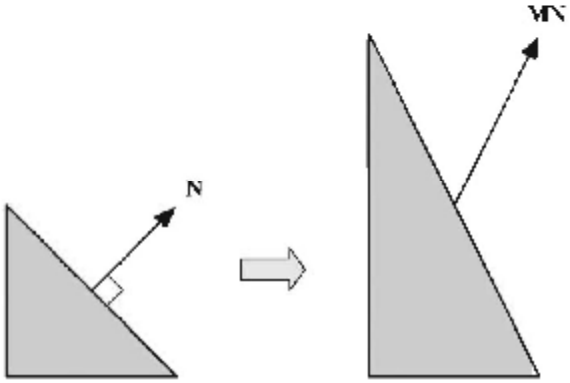
(1.3.21)

απαιτώντας να παρακολουθούμε το συστατικό πίνακα $\mathbf{M}_n\mathbf{M}_{n-1}$ καθώς επίσης και το συστατικό μεταφοράς $\mathbf{M}_n\mathbf{T}_{n-1} + \mathbf{T}_n$ σε κάθε στάδιο, όταν συνενώνεται ο n μετασχηματίζεται.

1.3.6 ΜΕΤΑΣΧΗΜΑΤΙΖΟΝΤΑΣ ΚΑΝΟΝΙΚΑ ΔΙΑΝΥΣΜΑΤΑ

Εκτός από τη θέση του στο χώρο, μια κορυφή που ανήκει σε ένα πολυγωνικό μοντέλο φέρει συνήθως επιπλέον πληροφορίες σχετικά με τον τρόπο με τον οποίο ταιριάζει στην περιβάλλουσα επιφάνεια. Συγκεκριμένα, μια κορυφή μπορεί να έχει εφαπτόμενο διάνυσμα και ένα κανονικό διάνυσμα που συνδέεται με αυτό. Όταν μεταφέρουμε ένα μοντέλο, πρέπει να μεταφέρουμε όχι μόνο τις θέσεις των κορυφών, αλλά και αυτά τα διανύσματα.

Τα εφαπτόμενα διανύσματα μπορούν συχνά να υπολογιστούν λαμβάνοντας τη διαφορά μεταξύ δύο μετασχηματισμένων σημείων. Αν **M** είναι ένας πίνακας 3 x 3 με τον οποίο μετασχηματίζουμε μια θέση κορυφής, τότε ο ίδιος πίνακας **M** μπορεί να χρησιμοποιηθεί για σωστό μετασχηματισμό του εφαπτόμενου διανύσματος σε αυτή την κορυφή. (Περιοριζόμαστε σε πίνακες 3 x 3 σε αυτό το τμήμα αφού οι εφαπτόμενες και οι κανονικές κατευθύνσεις δεν επηρεάζονται από τις μεταφράσεις). Πρέπει να ληφθεί μέριμνα για τη μετατροπή κανονικών διανυσμάτων. Η εικόνα 1.13 δείχνει τι μπορεί να συμβεί όταν χρησιμοποιείται μη ορθογώνιος πίνακας **M** για το μετασχηματισμό ενός κανονικού διανύσματος. Το μετασχηματισμένο normal μπορεί συχνά να καταλήγει σε κατεύθυνση που δεν είναι κάθετη προς τη μετασχηματισμένη επιφάνεια.



Εικόνα 1.13 Μετασχηματίζοντας ένα κανονικό διάνυσμα **N** με έναν μη ορθογώνιο πίνακα **M**

Δεδομένου ότι τα εφαπτόμενα και τα normals είναι κάθετα, το εφαπτόμενο διάνυσμα **T** και το κανονικό διάνυσμα **N** που σχετίζονται με μια κορυφή πρέπει να ικανοποιούν την εξίσωση $\mathbf{N} \cdot \mathbf{T} = 0$. Πρέπει επίσης να απαιτήσουμε αυτή η εξίσωση να ικανοποιηθεί από το μετασχηματισμένο εφαπτόμενο διάνυσμα **T'** και το μετασχηματισμένο κανονικό διάνυσμα **N'**. Δεδομένου ενός πίνακα μετασχηματισμού **M**, γνωρίζουμε ότι **T' = MT**. Θα θέλαμε να βρούμε τον πίνακα μετασχηματισμού **G** με το οποίο το διάνυσμα **N** θα πρέπει να μετασχηματιστεί έτσι ώστε

$$\mathbf{N'} \cdot \mathbf{T'} = (\mathbf{GN}) \cdot (\mathbf{MT}) = \mathbf{0}$$

Μια μικρή αλγεβρική διαχείριση μας δίνει

$$(\mathbf{GN}) \cdot (\mathbf{MT}) = (\mathbf{GN})^{\mathbf{T}} (\mathbf{MT}) = \mathbf{N}^{\mathbf{T}} \mathbf{G}^{\mathbf{T}} \mathbf{MT}.$$

Από $\mathbf{N}^{\mathbf{T}} \mathbf{T} = 0$, η εξίσωση $\mathbf{N}^{\mathbf{T}} \mathbf{G}^{\mathbf{T}} \mathbf{MT} = 0$ ικανοποιείται εάν $\mathbf{G}^{\mathbf{T}} \mathbf{M} = \mathbf{I}$. Συμπεραίνουμε λοιπόν ότι $\mathbf{G} = (\mathbf{M}^{-1})^{\mathbf{T}}$. Αυτό μας λέει ότι ένα κανονικό διάνυσμα μετασχηματίζεται σωστά χρησιμοποιώντας την αντίστροφη μεταφορά του πίνακα που χρησιμοποιείται για τη μετατροπή των σημείων. Τα διανύσματα που πρέπει να μετασχηματιστούν με αυτόν τον τρόπο ονομάζονται μεταφορικά διανύσματα και τα διανύσματα που μετασχηματίζονται με τον συνηθισμένο τρόπο χρησιμοποιώντας τον πίνακα **M** (όπως τα σημεία και τα εφαπτόμενα διανύσματα) ονομάζονται αντίθετα διανύσματα.

Εάν ο πίνακας **M** είναι ορθογώνιος, τότε $\mathbf{M}^{-1} = \mathbf{M}^{\mathbf{T}}$ και επομένως $(\mathbf{M}^{-1})^{\mathbf{T}} = \mathbf{M}$. Συνεπώς, μπορεί να αποφευχθεί η λειτουργία αντίστροφης μεταθέσεως που απαιτείται για τον μετασχηματισμό κανονικών διανυσμάτων όταν το M είναι γνωστό ότι είναι ορθογώνιο, όπως

συμβαίνει όταν το **M** είναι ίσο με έναν από τους πίνακες περιστροφής $\mathbf{R_x}$, $\mathbf{R_y}$, $\mathbf{R_z}$ ή $\mathbf{R_A}$.

2 THE GRAPHICS RENDERING PIPELINE

(Rodriguez)Οι διαφορές μεταξύ της συνήθους αρχιτεκτονικής CPU και μιας GPU είναι αρκετά μεγάλες ώστε να δικαιολογούν την αναφορά τους. Όταν προγραμματίζαμε εφαρμογές στο παρελθόν, γνωρίζαμε το βασικό υλικό: είχαμε CPU, ALU και μνήμη (τόσο προσωρινή όσο και για μαζική αποθήκευση) και ορισμένους τύπους συσκευών εισόδου / εξόδου (πληκτρολόγιο, οθόνη και ούτω καθεξής). Γνωρίζαμε επίσης ότι το πρόγραμμά μας θα έτρεχε διαδοχικά, μια εντολή μετά την άλλη (εκτός αν χρησιμοποιούσαμε multithreading). Κατά τον προγραμματισμό των shaders, αυτοί θα τρέχουν σε μια απομονωμένη μονάδα που ονομάζεται GPU, η οποία έχει μια πολύ διαφορετική αρχιτεκτονική από αυτή που έχουμε συνηθίσει.

Τώρα, η εφαρμογή μας θα τρέξει σε ένα μαζικό παράλληλο περιβάλλον. Οι συσκευές εισόδου / εξόδου είναι τελείως διαφορετικές. Δεν θα έχουμε άμεση πρόσβαση σε οποιοδήποτε είδος μνήμης, ούτε θα είναι δεδομένο για μας να το χρησιμοποιήσουμε σύμφωνα με τη θέλησή μας. Επίσης, το σύστημα θα δημιουργήσει το πρόγραμμά μας σε δεκάδες ή εκατοντάδες περιπτώσεις, σαν να έτρεχαν χρησιμοποιώντας εκατοντάδες πραγματικές διασυνδέσεις υλικού.

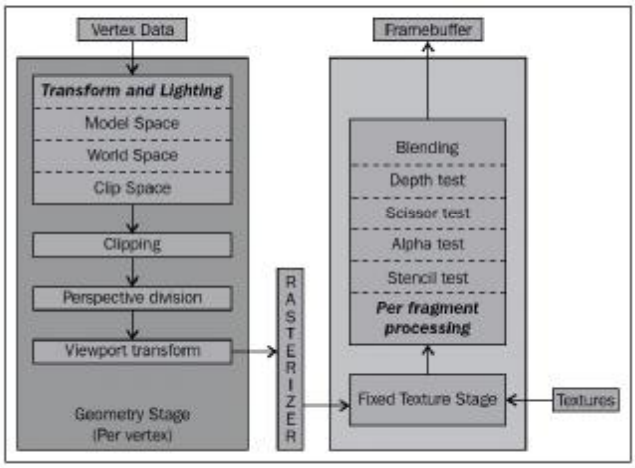
2.1 ΜΙΑ ΣΥΝΤΟΜΗ ΙΣΤΟΡΙΑ ΥΛΙΚΟΥ ΓΡΑΦΙΚΩΝ

Το υλικό γραφικών (που ονομάζεται επίσης κάρτα γραφικών ή GPU) δεν είναι μόνο μια δέσμη τρανζίστορ που λαμβάνει μερικές γενικές εντολές και δεδομένα εισόδου. Ενεργεί ως ένα CPU. Οι εντολές που δίδονται στο υλικό πρέπει να είναι συνεπείς και να έχουν σαφή και γνωστή σειρά σε κάθε στάδιο. Υπάρχουν επίσης απαιτήσεις δεδομένων για να δουλέψουμε όπως αναμενόταν (για παράδειγμα, δεν μπορούμε να χρησιμοποιήσουμε κορυφές ως είσοδο σε fragment shaders ή υφή ως έξοδο σε geometry shaders). Τα δεδομένα και οι εντολές πρέπει να ακολουθούν μια διαδρομή και πρέπει να περάσουν από ορισμένα στάδια και αυτό δεν μπορεί να αλλάξει.

Αυτή η διαδρομή ονομάζεται συνήθως σωλήνωση/αγωγός απόδοσης γραφικών. Ας το σκεφτούμε σαν ένα σωλήνα όπου εισάγουμε κάποια δεδομένα στο ένα άκρο - κορυφές, υφές, shaders - και αρχίζουν να ταξιδεύουν μέσω ορισμένων μικρών μηχανών που εκτελούν πολύ ακριβείς και συγκεκριμένες λειτουργίες στα δεδομένα και παράγουν την τελική έξοδο στο άλλο άκρο: Την τελική απόδοση(rendering).

Στα πρώτα OpenGL χρόνια, όπως αναφέρθηκε σε προηγούμενο κεφάλαιο, η σωλήνωση απόδοσης γραφικών ήταν τελείως σταθερή, πράγμα που σημαίνει ότι τα δεδομένα έπρεπε πάντοτε να περνούν από τις ίδιες μικρές μηχανές, που πάντα πραγματοποιούσαν τις ίδιες λειτουργίες, με την ίδια σειρά και δεν μπορούσε να παραλειφθεί καμία λειτουργία. Αυτές ήταν οι περίοδοι προ-shader (2002 και νωρίτερα).

Η ακόλουθη είναι μια απλοποιημένη αναπαράσταση της σταθερής σωλήνωσης, παρουσιάζοντας τα σημαντικότερα δομικά στοιχεία και τον τρόπο ροής των δεδομένων:



Εικόνα 2.1 Απλοποιημένη αναπαράσταση της σταθερής σωλήνωσης

Μεταξύ των ετών 2002 και 2004, υπήρχε κάποιο είδος προγραμματισμού εντός της GPU, αντικαθιστώντας ορισμένα από αυτά τα σταθερά στάδια. Ήταν οι πρώτοι shaders που οι προγραμματιστές γραφικών έπρεπε να κωδικοποιήσουν σε μια ψευδο-συναρμολογημένη γλώσσα, και ήταν σε πολύ ειδική πλατφόρμα. Στην πραγματικότητα, οι προγραμματιστές έπρεπε να κωδικοποιήσουν τουλάχιστον μία παραλλαγή shader για κάθε προμηθευτή υλικού γραφικών, επειδή δεν μοιράζονταν καν την ίδια γλώσσα συναρμολόγησης, αλλά τουλάχιστον μπορούσαν να αντικαταστήσουν μερικά από τα παλαιά στάδια σταθερής σωλήνωσης με μικρά χαμηλού επιπέδου προγράμματα. Παρ 'όλα αυτά, αυτή ήταν η αρχή της μεγαλύτερης επανάστασης στην ιστορία προγραμματισμού γραφικών σε πραγματικό χρόνο.

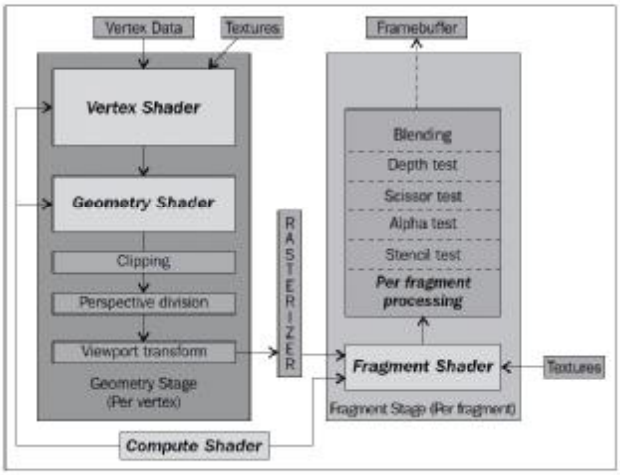
Ορισμένες εταιρείες παρείχαν στους προγραμματιστές άλλες λύσεις προγραμματισμού υψηλού επιπέδου, όπως το Cg (από τη NVidia) ή το HLSL (από τη Microsoft), αλλά οι λύσεις αυτές δεν ήταν για πολλαπλές πλατφόρμες. Το Cg χρησιμοποιήθηκε μόνο με GPUs NVidia και το HLSL ήταν μέρος του DIRECT3D.

Κατά τη διάρκεια του έτους 2004, ορισμένες εταιρείες συνειδητοποίησαν την ανάγκη για μια γλώσσα υψηλού επιπέδου shader, η οποία θα ήταν κοινή για διαφορετικές πλατφόρμες. Κάτι σαν ένα πρότυπο για τον προγραμματισμό shader. Ως εκ τούτου γεννήθηκε η γλώσσα σκίασης OpenGL (GLSL) και επέτρεψε στους προγραμματιστές να αντικαταστήσουν τις πολλαπλές διαδρομές του κώδικα του μεταφραστή συμβόλων με ένα μοναδικό (τουλάχιστον θεωρητικά, επειδή οι διαφορετικές GPU έχουν διαφορετικές δυνατότητες) C-like shader, κοινό για κάθε προμηθευτή υλικού.

Κατά το έτος αυτό, θα μπορούσαν να αντικατασταθούν μόνο δύο μέρη της σταθερής σωλήνωσης: η μονάδα επεξεργασίας των κορυφών(vertex processing unit), η οποία φρόντιζε για το μετασχηματισμό και το φωτισμό (T & L) και η μονάδα επεξεργασίας fragment(fragment processing unit) που ήταν υπεύθυνη για την εκχώρηση χρωμάτων σε εικονοστοιχεία. Αυτές οι νέες προγραμματιζόμενες μονάδες ονομαζόταν vertex shaders και

fragment shaders αντίστοιχα. Επίσης, άλλα δύο στάδια προστέθηκαν αργότερα. Οι geometry shaders και οι compute shaders προστέθηκαν στην επίσημη προδιαγραφή OpenGL το 2008 και το 2012 αντίστοιχα.

Το παρακάτω διάγραμμα δείχνει μια πτυχή της νέας προγραμματιζόμενης σωλήνωσης μετά την αλλαγή προγραμματισμού:



Εικόνα 2.2 Διάγραμμα για τη νέα προγραμματιζόμενη σωλήνωση

2.2 ΣΩΛΗΝΩΣΗ ΑΠΟΔΟΣΗΣ ΓΡΑΦΙΚΩΝ

Σύμφωνα με το διάγραμμα προγραμματιζόμενης σωλήνωσης, θα περιγράψουμε συνοπτικά το κομμάτι που διαβιβάζει τα δεδομένα για να εξηγήσει πώς μετασχηματίζεται σε κάθε στάδιο.

2.2.1 ΣΤΑΔΙΑ ΓΕΩΜΕΤΡΙΑΣ (ΛΕΙΤΟΥΡΓΙΕΣ ΑΝΑ ΚΟΡΥΦΗ)

Αυτό το κομμάτι των σταδίων επικεντρώνεται στη μετατροπή των δεδομένων κορυφών από την αρχική τους κατάσταση (σύστημα συντεταγμένων μοντέλου) στην τελική τους κατάσταση (σύστημα συντεταγμένων προβολής):

- **Δεδομένα κορυφής**: αυτά είναι τα δεδομένα εισόδου για ολόκληρη τη διαδικασία. Εδώ τροφοδοτούμε τον αγωγό με όλα τα διανυσματικά δεδομένα της γεωμετρίας μας: κορυφές, normals, δείκτες, tangents, binormals, συντεταγμένες υφής και ούτω καθεξής.

- **Υφές**: Όταν εμφανίστηκαν οι shaders, αυτή η νέα είσοδος για το στάδιο των κορυφών ήταν δυνατή. Εκτός από την απόδοση των χρωμάτων μας, οι υφές θα μπορούσαν να χρησιμεύσουν ως είσοδος σε vertex και geometry shaders, για παράδειγμα, να εκτοπίζει κορυφές σύμφωνα με τις τιμές που αποθηκεύονται σε μια υφή (Τεχνική μετατόπισης χαρτογράφησης).

- **Vertex Shader**: Αυτό το σύστημα είναι υπεύθυνο για τη μετατροπή των κορυφών από το τοπικό σύστημα συντεταγμένων τους στο χώρο κλιπ, εφαρμόζοντας τους κατάλληλους πίνακες μετασχηματισμού (μοντέλο, θέαση και προβολή).

- **Geometry Shader**: Θα μπορούσαν να δημιουργηθούν νέα πρωτόκολλα χρησιμοποιώντας αυτήν την ενότητα, με το αποτέλεσμα του vertex shader ως είσοδο.

- **Clipping**: Μόλις οι κορυφές του πρωτοκόλλου βρίσκονται στον λεγόμενο χώρο αποκοπής, είναι πιο εύκολο και υπολογιστικά φθηνότερο να αποσπόνται και να απορρίπτονται τα εξωτερικά τρίγωνα εδώ και όχι σε οποιοδήποτε άλλο χώρο.

- **Προοπτική διαίρεση**: Αυτή η ενέργεια μετατρέπει τον όγκο απεικόνισης (μια κολοβωμένη πυραμίδα, συνήθως αποκαλούμενη κόλουρο) σε έναν κανονικό και κανονικοποιημένο κύβο.

- **Μετασχηματισμός προβολής**: Η κοντινότερη επιφάνεια του όγκου clipping(ο κανονικοποιημένος κύβος) μεταφέρεται και κλιμακώνεται στις συντεταγμένες του παραθύρου προβολής. Αυτό σημαίνει ότι οι συντεταγμένες θα αντιστοιχιστούν στο παράθυρο προβολής (συνήθως στην οθόνη μας ή στο παράθυρό μας).

- **Τα δεδομένα μεταβιβάζονται στον rasterizer**: Αυτό είναι το στάδιο που μας μετατρέπει διανυσματικά δεδομένα (οι κορυφές του πρωτοκόλλου) σε μια διακριτή αναπαράσταση (το framebuffer) προς επεξεργασία σε περαιτέρω βήματα.

2.2.2 ΣΤΑΔΙΑ FRAGMENT (ΛΕΙΤΟΥΡΓΙΕΣ ΑΝΑ ΤΕΜΑΧΙΟ)

Εδώ τα διανυσματικά μας δεδομένα μετασχηματίζονται σε διακεκριμένα δεδομένα, έτοιμα να ραστεροποιηθούν. Τα στάδια εντός των ελέγχων superblock δείχνουν ότι διακριτά δεδομένα τελικά θα παρουσιαστούν :

- **Fragment Shader**: Αυτό είναι το στάδιο όπου η υφή, τα χρώματα και τα φώτα υπολογίζονται, εφαρμόζονται και συνδυάζονται για να σχηματίσουν ένα fragment.

- **Post Fragment Processing**: Αυτό είναι το στάδιο στο οποίο πραγματοποιείται η ανάμειξη, οι δοκιμές βάθους, οι δοκιμές scissor, οι δοκιμές άλφα και ούτω καθεξής. Τα fragments συνδυάζονται, ελέγχονται και απορρίπτονται σε αυτό το στάδιο και εκείνα που τελικά περνούν, γράφονται στο framebuffer.

2.2.3 ΕΞΩΤΕΡΙΚΑ ΣΤΑΔΙΑ

Έξω από τα μεγάλα μπλοκ ανά κορυφή και ανά fragment βρίσκεται το στάδιο compute shader. Αυτό το στάδιο μπορεί να γραφτεί για να επηρεάσει οποιοδήποτε άλλο προγραμματιζόμενο τμήμα του αγωγού.

2.3 ΔΙΑΦΟΡΕΣ ΜΕΤΑΞΥ ΣΤΑΘΕΡΩΝ ΚΑΙ ΠΡΟΓΡΑΜΜΑΤΙΖΟΜΕΝΩΝ ΣΧΕΔΙΩΝ

Αξίζει να κατανοηθεί η σταθερή σωλήνωση, επειδή η προγραμματιζόμενη σωλήνωση βασίζεται σε μεγάλο βαθμό σε αυτή. Οι Shaders αντικαθιστούν μόνο μερικές καλά καθορισμένες ενότητες που υπήρχαν στο παρελθόν με σταθερό τρόπο, οπότε η έννοια της "σωλήνωσης" δεν άλλαξε στην πραγματικότητα.

Στην περίπτωση των vertex shaders, αντικαθιστούν ολόκληρη τη μονάδα μετασχηματισμού και φωτισμού. Τώρα πρέπει να γράψουμε ένα πρόγραμμα που μπορεί να εκτελέσει ισοδύναμα καθήκοντα. Μέσα στον vertex shader, μπορούμε να εκτελέσουμε τους υπολογισμούς που θα χρειαστούμε για τους σκοπούς μας, αλλά υπάρχει μια ελάχιστη απαίτηση. Για να μην σπάσει η σωλήνωση, η έξοδος του shader πρέπει να τροφοδοτεί την είσοδο της επόμενης μονάδας. Μπορούμε να το επιτύχουμε με τον υπολογισμό της θέσης κορυφής σε συντεταγμένες clipping και γράφοντάς τη για το επόμενο στάδιο.

Όσον αφορά τους fragment shaders , αντικαθιστούν τα στάδια σταθερής υφής. Στο παρελθόν, αυτή η ενότητα ασχολήθηκε με το πώς ένα fragment δημιουργήθηκε συνδυάζοντας τις υφές με πολύ περιορισμένο τρόπο. Επί του παρόντος, το τελικό αποτέλεσμα ενός τμήματος shader είναι ένα fragment. Όπως είπαμε προηγουμένως, ένα fragment είναι υποψήφιο προς ένα εικονοστοιχείο, οπότε στην πιο απλή του μορφή είναι απλά ένα χρώμα RGBA. Για να συνδέσουμε το fragment shader με τις παρακάτω ενότητες της σωλήνωσης, θα πρέπει να παράγουμε αυτό το χρώμα, αλλά μπορούμε να το υπολογίσουμε όπως θέλουμε.

Όταν ο fragment shader παράγει ένα χρώμα, άλλα δεδομένα σχετίζονται επίσης με αυτό, κυρίως η θέση του και το βάθος του, έτσι ώστε περαιτέρω δοκιμές όπως οι δοκιμές depth ή scissor να μπορούν να γίνουν κατευθείαν. Αφού επεξεργαστούν όλα τα fragments για μια τρέχουσα θέση ράστερ, το χρώμα που παραμένει είναι αυτό που συνήθως ονομάζεται pixel.

Προαιρετικά, μπορούμε να ορίσουμε δύο επιπλέον μονάδες που δεν υπήρχαν στη σταθερή σωλήνωση πριν:

- **Geometry Shader**: Η μονάδα αυτή τοποθετείται μετά τον vertex shader, αλλά πριν συμβεί η αποκοπή(clipping). Η ευθύνη αυτής της μονάδας είναι να εκπέμπει νέα πρωτόκολλα (όχι κορυφές!) με βάση τις εισόδους.

- **Compute Shader**: Αυτή είναι μια συμπληρωματική μονάδα. Με κάποιο τρόπο, αυτό είναι εντελώς διαφορετικό από τα άλλα shaders επειδή επηρεάζει ολόκληρη τη

σωλήνωση παγκοσμίως. Ο κύριος σκοπός του είναι να παρέχει μια μέθοδο για γενικό GPGPU (Υπολογισμός γενικής χρήσης σε μονάδες GPU). Είναι όπως το OpenCL, αλλά πιο βολικό για προγραμματιστές γραφικών επειδή είναι πλήρως ενσωματωμένο σε ολόκληρη τη σωλήνωση. Ως παραδείγματα χρήσης στα γραφικά, θα μπορούσαν να χρησιμοποιηθούν για μετασχηματισμούς εικόνας ή για αναβολή rendering με πιο αποτελεσματικό τρόπο από το OpenCL.

2.4 ΤΥΠΟΙ SHADERS

Οι Vertex και Fragment είναι οι σημαντικότεροι shaders σε ολόκληρη τη σωλήνωση, επειδή εκθέτουν την ατόφια βασική λειτουργικότητα της GPU. Με τους Vertex Shaders, μπορούμε να υπολογίσουμε τη γεωμετρία του αντικειμένου που πρόκειται να εκτελέσουμε καθώς και άλλα σημαντικά στοιχεία, όπως η κάμερα της σκηνής, η προβολή ή ο τρόπος με τον οποίο κόβεται η γεωμετρία. Με τους Fragment Shaders, μπορούμε να ελέγξουμε τον τρόπο εμφάνισης της γεωμετρίας μας στην οθόνη: χρώματα, φωτισμό, υφές και ούτω καθεξής.

Όπως βλέπουμε, με μόνο Vertex και Fragment Shaders , μπορούμε να ελέγχουμε σχεδόν τα πάντα στη διαδικασία απόδοσης, αλλά υπάρχει περιθώριο για μεγαλύτερη βελτίωση στη μηχανή OpenGL.

Ας δώσουμε ένα παράδειγμα: ας υποθέσουμε ότι επεξεργαζόμαστε πρωτόκολλα σημείων με ένα σύνθετο vertex shader. Χρησιμοποιώντας αυτές τις επεξεργασμένες κορυφές, μπορούμε να χρησιμοποιήσουμε έναν geometry shader για να δημιουργήσουμε αυθαίρετα διαμορφωμένα πρωτόκολλα (για παράδειγμα, τετράγωνα) χρησιμοποιώντας τα σημεία ως κέντρο του τετραγώνου. Τότε μπορούμε να χρησιμοποιήσουμε αυτά τα quads(τετράγωνα) για ένα σύστημα σωματιδίων.

Κατά τη διάρκεια αυτής της διαδικασίας έχουμε αποθηκεύσει το εύρος ζώνης γιατί έχουμε στείλει σημεία αντί για τετράγωνα που έχουν τέσσερις φορές περισσότερες κορυφές και ισχύ επεξεργασίας επειδή, αφού μετασχηματίσαμε τα σημεία, οι άλλες τέσσερις κορυφές βρίσκονται ήδη στον ίδιο χώρο, έτσι ώστε να μετασχηματίσουμε μια κορυφή με ένα σύνθετο shader αντί για τέσσερα.

Σε αντίθεση με τους shaders vertex και fragment(είναι υποχρεωτικό να έχουμε ένα από κάθε είδος για να ολοκληρωθεί η σωλήνωση) ο geometry shader είναι μόνο προαιρετικός. Έτσι, αν δεν θέλουμε να δημιουργήσουμε μια νέα γεωμετρία μετά την εκτέλεση του vertex shader, απλά δε θα συνδέσουμε έναν geometry shader στην εφαρμογή μας και τα αποτελέσματα του vertex shader θα περάσουν αμετάβλητα στο στάδιο αποκοπής(clipping), το οποίο είναι απόλυτα εντάξει.

Το στάδιο compute shader ήταν η τελευταία προσθήκη στη σωλήνωση. Είναι επίσης προαιρετικό, όπως και ο geometry shader, και προορίζεται για γενικούς υπολογισμούς.

Εντός του αγωγού, μπορεί να υπάρχουν κάποιοι από τους ακόλουθους shaders: vertex

shaders, fragment shaders, geometry shaders , tesselation shaders (που προορίζονται για υποδιαίρεση τριγωνικών πλεγμάτων σε κίνηση) και compute shaders. Η OpenGL εξελίσσεται καθημερινά, οπότε δεν θα αποτελέσει έκπληξη αν εμφανιστούν άλλες κλάσεις shader και αλλάζουν τη διάταξη του αγωγού κατά διαστήματα.

Πριν προχωρήσουμε βαθύτερα στο θέμα, υπάρχει μια σημαντική έννοια για την οποία πρέπει να μιλήσουμε. Την έννοια ενός προγράμματος shader. Ένα πρόγραμμα shader δεν είναι τίποτα άλλο παρά μια διαμόρφωση του αγωγού που βρίσκεται σε λειτουργία. Αυτό σημαίνει ότι τουλάχιστον ένας vertex shader και ένας fragment shader πρέπει να έχουν καταρτιστεί χωρίς σφάλματα και να συνδεθούν μεταξύ τους. Όσον αφορά τους geometry και compute shaders, θα μπορούσαν επίσης να αποτελέσουν μέρος ενός προγράμματος, να συντάσσονται και να συνδέονται μαζί με τους άλλους δύο shaders στο ίδιο πρόγραμμα shader.

2.4.1 VERTEX SHADERS

Για να λάβουμε τις συντεταγμένες του 3D μοντέλου μας και να τις μετασχηματίσουμε στο χώρο κλιπ, συνήθως εφαρμόζουμε τους πίνακες μοντέλου(model), οπτικής(view) και προβολής(projection) στις κορυφές. Επίσης, μπορούμε να εκτελέσουμε οποιοδήποτε άλλο τύπο μετασχηματισμού δεδομένων, όπως εφαρμογή θορύβου στις θέσεις για την μετατόπιση ψευδοτυχαίων, υπολογισμό normals, υπολογισμό συντεταγμένων υφής, υπολογισμό χρωμάτων κορυφών, προετοιμασία δεδομένων για κανονικό shader χαρτογράφησης, και ούτω καθεξής.

Μπορούμε να κάνουμε πολλά περισσότερα με αυτό το shader. Ωστόσο, η πιο σημαντική πτυχή του είναι να παράσχει τις θέσεις των κορυφών για να ενώσει τις συντεταγμένες και να μας οδηγήσει στο επόμενο στάδιο.

Ένας vertex shader είναι ένα κομμάτι κώδικα που εκτελείται στους επεξεργαστές GPU και εκτελείται μία και μόνο μία φορά για κάθε κορυφή που στέλνουμε στην κάρτα γραφικών. Έτσι, αν έχουμε ένα μοντέλο 3D με 1000 κορυφές, το Vertex shader θα εκτελεστεί 1000 φορές, οπότε θυμόμαστε να κρατάτε τους υπολογισμούς μας πάντα απλούς.

2.4.2 FRAGMENT SHADERS

Οι fragment shaders είναι υπεύθυνοι για τη ζωγραφική/χρωματισμό της περιοχής κάθε πρωτοκόλλου. Η ελάχιστη εργασία για ένα fragment shader είναι η έξοδος ενός χρώματος RGBA. Μπορούμε να υπολογίσουμε αυτό το χρώμα με οποιονδήποτε τρόπο: διαδικαστικά, από υφές, ή χρησιμοποιώντας τα δεδομένα εξόδου vertex shader. Αλλά στο τέλος, θα πρέπει να εξάγουμε τουλάχιστον ένα χρώμα στο framebuffer.

Το μοντέλο εκτέλεσης ενός fragment shader είναι σαν αυτό του vertex shader. Ένας fragment shader είναι ένα κομμάτι κώδικα που εκτελείται μία φορά, και μόνο μία φορά, ανά fragment. Ας επεξεργαστούμε αυτό λίγο. Ας υποθέσουμε ότι έχουμε μια οθόνη με μέγεθος

1.024 x 768. Αυτή η οθόνη περιέχει 786.432 pixels. Τώρα υποθέτουμε ότι ζωγραφίζουμε ένα τετράγωνο που καλύπτει ακριβώς ολόκληρη την οθόνη (επίσης γνωστή ως quad πλήρους οθόνης). Αυτό σημαίνει ότι ο fragment shader θα εκτελεστεί 786.432 φορές, αλλά η πραγματικότητα είναι χειρότερη. Τι γίνεται αν ζωγραφίσουμε πολλά τετράγωνα πλήρους οθόνης (κάτι φυσιολογικό όταν κάνουμε μετασχηματισμούς shaders όπως θαμπάδα κινήσεων, λάμψη ή απόφραξη περιβάλλοντος χώρου οθόνης) ή απλά πολλά τρίγωνα που επικαλύπτονται στην οθόνη; Κάθε φορά που ζωγραφίζουμε ένα τρίγωνο στην οθόνη, όλη η περιοχή του πρέπει να είναι ραστεροποιημένη, έτσι πρέπει να υπολογίζονται όλα τα fragments του τριγώνου. Στην πραγματικότητα, ένας fragment shader εκτελείται εκατομμύρια φορές. Η βελτιστοποίηση σε ένα fragment shader είναι πιο κρίσιμη από ό, τι στους vertex shaders.

2.4.3 GEOMETRY SHADERS

Το στάδιο του geometry shader είναι υπεύθυνο για τη δημιουργία νέων ρενταρισμένων πρωτοκόλλων που διαχωρίζονται από την έξοδο του vertex shader. Ένας geometry shader εκτελείται μία φορά ανά πρωτόκολλο, που είναι, στη χειρότερη περίπτωση (όταν χρησιμοποιείται για την εκπομπή πρωτοκόλλων σημείων), το ίδιο με το vertex shader. Το καλύτερο σενάριο είναι όταν χρησιμοποιείται για να εκπέμπει τρίγωνα, γιατί μόνο τότε θα εκτελεστεί τρεις φορές λιγότερο από το vertex shader, αλλά αυτή η πολυπλοκότητα είναι σχετική. Παρόλο που η εκτέλεση του geometry shader θα μπορούσε να είναι φτηνή, αυξάνει πάντοτε την πολυπλοκότητα της σκηνής, και αυτό πάντα μεταφράζεται σε περισσότερο υπολογιστικό χρόνο που δαπανάται από τη GPU για να αποδώσει τη σκηνή.

2.4.4 COMPUTE SHADERS

Αυτό το ειδικό είδος shader δεν σχετίζεται άμεσα με ένα συγκεκριμένο τμήμα του αγωγού. Μπορούν να γραφτούν(οι compute shaders) για να επηρεάσουν τους shaders vertex, fragment ή geometry.

Καθώς οι compute shaders βρίσκονται με κάποιο τρόπο έξω από τον αγωγό, δεν έχουν τους ίδιους περιορισμούς με τους άλλους τύπους shaders. Αυτό τους καθιστά ιδανικούς για γενικούς υπολογισμούς. Οι compute shaders είναι λιγότερο συγκεκριμένοι, αλλά έχουν το πλεονέκτημα ότι έχουν πρόσβαση σε όλες τις λειτουργίες (`matrix`, λειτουργίες `advanced texture` κ.ο.κ.) και τύπους δεδομένων (`vectors`, `matrices`, όλες τις μορφές υφής και `vertex buffers`) που υπάρχουν στο GLSL, ενώ άλλες λύσεις GPGPU, όπως το OpenCL ή το CUDA, έχουν τους δικούς τους ειδικούς τύπους δεδομένων και δεν ταιριάζουν εύκολα με τον αγωγό απόδοσης.

3 ΓΕΝΙΚΑ ΣΤΟΙΧΕΙΑ ΓΙΑ ΤΗΝ OPEN GL

3.1 ΤΙ ΕΙΝΑΙ ΤΟ ΣΥΣΤΗΜΑ ΓΡΑΦΙΚΩΝ OPENGL;

(Mark Segal, September 23, 2008)Η OpenGL (για "Open Graphics Library") είναι μια διασύνδεση λογισμικού στο υλικό των γραφικών. Η διεπαφή αποτελείται από ένα σύνολο αρκετών εκατοντάδων διαδικασιών και λειτουργιών που επιτρέπουν σε έναν προγραμματιστή να καθορίζει τα αντικείμενα και τις λειτουργίες που εμπλέκονται στην παραγωγή γραφικών εικόνων υψηλής ποιότητας, ιδιαίτερα έγχρωμες εικόνες τρισδιάστατων αντικειμένων.

Το μεγαλύτερο μέρος της OpenGL απαιτεί το υλικό των γραφικών να περιέχει ένα frame buffer. Πολλές κλήσεις OpenGL σχετίζονται με την σχεδίαση αντικειμένων όπως σημεία, γραμμές, πολύγωνα και bitmap, αλλά ο τρόπος με τον οποίο συμβαίνουν μερικά από αυτά τα σχέδια (όπως όταν ενεργοποιείται η antialiasing ή η υφή), βασίζεται στην ύπαρξη ενός Framebuffer. Επιπλέον, ένα κομμάτι της OpenGL ασχολείται ειδικά με τη διαχείριση του frame buffer.

Για τον προγραμματιστή, η OpenGL είναι ένα σύνολο εντολών που επιτρέπουν τον προσδιορισμό γεωμετρικών αντικειμένων σε δύο ή τρεις διαστάσεις, μαζί με εντολές που ελέγχουν τον τρόπο με τον οποίο τα αντικείμενα αυτά “ρεντάρονται” στο framebuffer. Ως επί το πλείστον, η OpenGL παρέχει μια διεπαφή άμεσης λειτουργίας, πράγμα που σημαίνει ότι ο προσδιορισμός ενός αντικειμένου προκαλεί την έξοδό του.

Ένα τυπικό πρόγραμμα που χρησιμοποιεί η OpenGL ξεκινά με κλήσεις για να ανοίξει ένα παράθυρο στο framebuffer στο οποίο θα τραβήξει το πρόγραμμα. Στη συνέχεια, πραγματοποιούνται κλήσεις για την κατανομή ενός περιβάλλοντος GL και τη συσχέτισή του με το παράθυρο. Μόλις κατανεμηθεί ένα πλαίσιο GL, ο προγραμματιστής είναι ελεύθερος να εκδώσει εντολές OpenGL. Μερικές κλήσεις χρησιμοποιούνται για την σχεδίαση απλών γεωμετρικών αντικειμένων (σημεία, γραμμικά τμήματα και πολύγωνα), ενώ άλλα επηρεάζουν την απόδοση αυτών των primitive στοιχείων, συμπεριλαμβανομένου του τρόπου με τον οποίο είναι φωτισμένα ή χρωματισμένα και τον τρόπο χαρτογράφησης τους από το μοντέλο δύο ή τριών διαστάσεων του χώρου προς την δισδιάστατη οθόνη. Υπάρχουν επίσης κλήσεις για άμεσο έλεγχο του framebuffer, όπως ανάγνωση και εγγραφή εικονοστοιχείων.

3.2 ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ ΓΙΑ ΤΗ ΛΕΙΤΟΥΡΓΙΑ ΤΟΥ OPENGL

Η OpenGL ασχολείται μόνο με το rendering σε framebuffer (και τις τιμές ανάγνωσης που είναι αποθηκευμένες σε αυτό το framebuffer). Δεν υπάρχει υποστήριξη για άλλες περιφερειακές συσκευές που συνδέονται με το υλικό των γραφικών, όπως ποντίκια και πληκτρολόγια. Οι προγραμματιστές θα πρέπει να βασίζονται σε άλλους μηχανισμούς για την απόκτηση δεδομένων από τους χρήστες.

Η GL σχεδιάζει πρωτόκολλα που υπόκεινται σε διάφορες επιλογές. Κάθε πρωτόκολλο είναι ένα ορθογώνιο σημείο, τμήμα γραμμής, πολύγωνο ή εικονοστοιχείο. Κάθε λειτουργία μπορεί να αλλάξει ανεξάρτητα. Η ρύθμιση ενός δεν επηρεάζει τις ρυθμίσεις άλλων (αν και πολλές λειτουργίες μπορεί να αλληλεπιδρούν για να καθορίσουν τι τελικά καταλήγει στο

framebuffer). Οι τρόποι λειτουργίας ρυθμίζονται, καθορίζονται πρωτόκολλα και άλλες λειτουργίες GL που περιγράφονται με την αποστολή εντολών με τη μορφή κλήσεων λειτουργίας ή διαδικασίας.

Τα πρωτόκολλα ορίζονται από μια ομάδα από μία ή περισσότερες κορυφές. Μια κορυφή ορίζει ένα σημείο, ένα τελικό σημείο μιας άκρης ή μια γωνία ενός πολύγωνου όπου συναντώνται δύο άκρες. Τα δεδομένα (που αποτελούνται από συντεταγμένες θέσης, χρώματα, normals και συντεταγμένες υφής) συνδέονται με μια κορυφή και κάθε κορυφή επεξεργάζεται ανεξάρτητα, με τη σειρά και με τον ίδιο τρόπο. Η μόνη εξαίρεση σε αυτόν τον κανόνα είναι εάν η ομάδα των κορυφών πρέπει να περικοπεί έτσι ώστε το ενδεικνυόμενο πρωτόκολλο να προσαρμόζεται εντός μιας καθορισμένης περιοχής. Στην περίπτωση αυτή τα δεδομένα των κορυφών μπορούν να τροποποιηθούν και να δημιουργηθούν νέες κορυφές. Ο τύπος αποκοπής(clipping) εξαρτάται από το πρωτόκολλο που αντιπροσωπεύει η ομάδα κορυφών.

Οι εντολές επεξεργάζονται πάντοτε με τη σειρά που λαμβάνονται, αν και μπορεί να υπάρξει μια απροσδιόριστη καθυστέρηση πριν πραγματοποιηθούν οι επιδράσεις μιας εντολής. Αυτό σημαίνει, για παράδειγμα, ότι ένα πρωτόκολλο πρέπει να σχεδιαστεί ακριβώς πριν από κάθε μεταγενέστερο που μπορεί να επηρεάσει το framebuffer. Σημαίνει επίσης ότι οι διαδικασίες ερωτήσεων και ανάγνωσης εικονοστοιχείων επιστρέφουν σε κατάσταση συμβατή με την πλήρη εκτέλεση όλων των εντολών GL που είχαν προηγουμένως ζητηθεί. Γενικά, οι επιδράσεις μιας εντολής GL είτε στις λειτουργίες GL είτε στο framebuffer πρέπει να είναι πλήρεις πριν από οποιαδήποτε μεταγενέστερη εντολή μπορεί να έχει τέτοια αποτελέσματα.

Στη GL, η δέσμευση δεδομένων πραγματοποιείται κατά την κλήση. Αυτό σημαίνει ότι τα δεδομένα που διαβιβάζονται σε μια εντολή ερμηνεύονται όταν λαμβάνεται αυτή η εντολή. Ακόμη και αν η εντολή απαιτεί δείκτη δεδομένων, τα δεδομένα αυτά ερμηνεύονται όταν πραγματοποιείται η κλήση και οι τυχόν μεταγενέστερες αλλαγές στα δεδομένα δεν έχουν επίδραση στη GL (εκτός αν χρησιμοποιείται ο ίδιος δείκτης σε μια επόμενη εντολή).

Η GL παρέχει άμεσο έλεγχο στις θεμελιώδεις λειτουργίες των γραφικών 3D και 2D. Αυτό περιλαμβάνει εξειδίκευση τέτοιων παραμέτρων, όπως πίνακες μετασχηματισμού, συντελεστές εξισώσεων φωτισμού, μέθοδοι antialiasing και χειριστές ενημερώσεων εικονοστοιχείων. Δεν παρέχει μέσο για την περιγραφή ή τη μοντελοποίηση σύνθετων γεωμετρικών αντικειμένων. Ένας άλλος τρόπος για να περιγράψουμε αυτή την κατάσταση είναι να πούμε ότι η GL παρέχει μηχανισμούς για να περιγράψει πόσο σύνθετα γεωμετρικά αντικείμενα πρόκειται να παραχθούν αντί για μηχανισμούς για να περιγράψουν τα ίδια τα σύνθετα αντικείμενα.

Το μοντέλο για την ερμηνεία των εντολών GL είναι client-server. Δηλαδή, ένα πρόγραμμα (ο client) εκδίδει εντολές και αυτές οι εντολές ερμηνεύονται και επεξεργάζονται από την GL (τον server). Ο server μπορεί να λειτουργεί ή να μην λειτουργεί στον ίδιο υπολογιστή με τον client. Με αυτή την έννοια, η GL είναι "διαφανής στο δίκτυο". Ένας server μπορεί να διατηρεί ένα πλήθος GL πλαισίων, καθένα από τα οποία είναι μια encapsulation(ενθυλάκωση) της τρέχουσας κατάστασης GL. Ένας client μπορεί να επιλέξει να συνδεθεί με οποιοδήποτε από αυτά τα περιβάλλοντα. Όταν το πρόγραμμα δεν είναι συνδεδεμένο με ένα περιβάλλον, οδηγεί σε απροσδιόριστη συμπεριφορά.

Τα αποτελέσματα των εντολών GL στο framebuffer τελικά ελέγχονται από το σύστημα παραθύρων που διαθέτει τους πόρους framebuffer. Είναι το σύστημα παραθύρων που καθορίζει ποια τμήματα του framebuffer της GL μπορεί να έχουν πρόσβαση σε οποιαδήποτε δεδομένη στιγμή και ότι επικοινωνούν με τη GL για το πως αυτά τα τμήματα είναι δομημένα. Επομένως, δεν υπάρχουν εντολές GL για να ρυθμίσουμε το framebuffer ή να αρχικοποιήσουμε τη GL. Ομοίως, η προβολή των περιεχομένων framebuffer σε μια οθόνη CRT (συμπεριλαμβανομένης της μετατροπής των επιμέρους τιμών framebuffer με τέτοιες τεχνικές όπως η διόρθωση γάμμα) δεν αντιμετωπίζεται από τη GL. Η ρύθμιση Framebuffer πραγματοποιείται εκτός της GL σε συνδυασμό με το σύστημα παραθύρων. Η αρχικοποίηση ενός πλαισίου GL συμβαίνει όταν το σύστημα παράθυρου διαθέτει ένα παράθυρο για GL rendering.

Η GL έχει σχεδιαστεί για να λειτουργεί σε μια σειρά πλατφορμών γραφικών με ποικίλες δυνατότητες γραφικών και απόδοση. Για να ικανοποιήσουμε αυτήν την ποικιλία, καθορίζουμε την ιδανική συμπεριφορά αντί της πραγματικής συμπεριφοράς για ορισμένες λειτουργίες GL. Σε περιπτώσεις όπου επιτρέπεται απόκλιση από το ιδανικό, καθορίζουμε επίσης τους κανόνες στους οποίους πρέπει να συμμορφώνεται μια εφαρμογή, προκειμένου να προσεγγίσει χρήσιμα την ιδανική συμπεριφορά. Αυτό επιτρέπει παραλλαγή στη συμπεριφορά GL και συνεπάγεται ότι δύο διαφορετικές υλοποιήσεις GL μπορεί να μην συμφωνούν pixel με pixel όταν παρουσιάζονται με την ίδια είσοδο ακόμη και όταν τρέχουν σε όμοιες διαμορφώσεις framebuffer.

Τέλος, τα ονόματα των εντολών, οι σταθερές και οι τύποι έχουν προκαθοριστεί στη GL (από gl, GL_ και GL, αντίστοιχα στο C) για να μειώσουν τις συγκρούσεις ονόματος με άλλα πακέτα.

3.3 ΒΑΣΙΚΗ ΛΕΙΤΟΥΡΓΙΑ GL

Το σχήμα 3.1 δείχνει ένα σχηματικό διάγραμμα της GL. Οι εντολές εισέρχονται στη GL στην αριστερή πλευρά. Ορισμένες εντολές καθορίζουν τα γεωμετρικά αντικείμενα που πρέπει να σχεδιαστούν, ενώ άλλες ελέγχουν τον τρόπο χειρισμού των αντικειμένων από τα διάφορα στάδια. Οι περισσότερες εντολές μπορούν να συσσωρευτούν σε μια λίστα εμφανίσεων(display list) για επεξεργασία αργότερα από τη GL. Διαφορετικά, οι εντολές αποστέλλονται αποτελεσματικά μέσω ενός αγωγού επεξεργασίας.

Letter	Corresponding GL Type
b	byte
s	short
i	int
f	float
d	double
ub	ubyte
us	ushort
ui	uint

Πίνακας 3.1: Αντιστοίχιση των χαρακτήρων επιθέτησης εντολών στους τύπους παραδειγμάτων GL.

Το πρώτο στάδιο παρέχει ένα αποτελεσματικό μέσο για την προσέγγιση της καμπύλης και της γεωμετρίας της επιφάνειας αξιολογώντας τις πολυωνυμικες λειτουργίες των τιμών εισόδου. Το επόμενο στάδιο λειτουργεί σε γεωμετρικά πρωτόκολλα που περιγράφονται από κορυφές: σημεία, τμήματα γραμμής και πολύγωνα. Σε αυτό το στάδιο οι κορυφές μετασχηματίζονται και φωτίζονται και τα πρωτόκολλα κόβονται σε έναν όγκο προβολής κατά την προετοιμασία για το επόμενο στάδιο της ραστεροποίησης. Ο rasterizer παράγει μια σειρά διευθύνσεων και τιμών του framebuffer χρησιμοποιώντας μια δισδιάστατη περιγραφή ενός σημείου, τμήματος γραμμής ή πολυγώνου. Κάθε fragment που παράγεται κατ 'αυτόν τον τρόπο τροφοδοτείται στο επόμενο στάδιο το οποίο εκτελεί λειτουργίες επί μεμονωμένων fragment προτού τελικά μεταβάλλουν το framebuffer. Αυτές οι λειτουργίες περιλαμβάνουν ενημερώσεις υπό όρους στο framebuffer με βάση τις εισερχόμενες και προηγουμένως αποθηκευμένες τιμές βάθους (για την πραγματοποίηση buffering βάθους), την ανάμειξη των εισερχόμενων χρωμάτων fragment με τα αποθηκευμένα χρώματα, καθώς και την κάλυψη και άλλες λογικές λειτουργίες σε τιμές fragment.

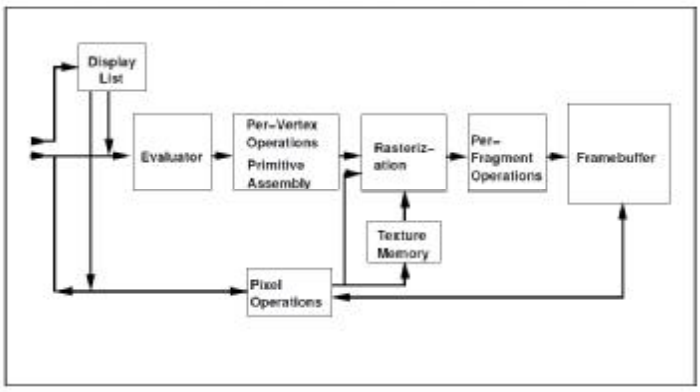
Τέλος, υπάρχει ένας τρόπος να παρακάμψουμε το τμήμα επεξεργασίας των κορυφών του αγωγού για να στείλουμε ένα μπλοκ από fragment απευθείας στις λειτουργίες μεμονωμένων fragment, προκαλώντας τελικά ένα μπλοκ εικονοστοιχείων να γραφτούν στο framebuffer. Οι τιμές μπορούν επίσης να διαβαστούν από το framebuffer ή να αντιγραφούν από ένα τμήμα του framebuffer σε ένα άλλο. Αυτές οι μεταφορές μπορεί να περιλαμβάνουν κάποιο τύπο αποκωδικοποίησης ή κωδικοποίησης.

Αυτή η διαδικασία χρησιμοποιείται μόνο ως εργαλείο για την περιγραφή της GL, όχι ως αυστηρός κανόνας για τον τρόπο με τον οποίο υλοποιείται η GL και την παρουσιάζουμε μόνο ως μέσο οργάνωσης των διαφόρων λειτουργιών της GL. Αντικείμενα όπως καμπύλες

επιφάνειες, για παράδειγμα, μπορούν να μετασχηματιστούν πριν μετατραπούν σε πολύγωνα.

GL Type	Ελάχιστος αριθμός Bits	Περιγραφή
boolean	1	Boolean
byte ακέραιο	8	Υπογεγραμμένο 2 συμπληρωματικό δυαδικό
ubyte	8	Μη υπογεγραμμένο δυαδικό ακέραιο
short ακέραιο	16	Υπογεγραμμένο 2 συμπληρωματικό δυαδικό
ushort	16	Μη υπογεγραμμένο δυαδικό ακέραιο
int ακέραιο	32	Υπογεγραμμένο 2 συμπληρωματικό δυαδικό
uint	32	Μη υπογεγραμμένο δυαδικό ακέραιο
sizei αριθμού	32	Μη αρνητικό μέγεθος δυαδικού ακέραιου
enum	32	Απαριθμημένη δυαδική τιμή ακέραιου αριθμού
bitfield	32	Πεδίο bit
float	32	Τιμή κινητής υποδιαστολής
clampd 1]	32	Η τιμή κινητής υποδιαστολής συσφίγγεται στο [0,
double στο[0, 1]	64	Η τιμή κινητής υποδιαστολής συσφίγγεται

Πίνακας 3.2: Τύποι δεδομένων GL. Οι τύποι GL δεν είναι τύποι C. Έτσι, για παράδειγμα, ο GL τύπος int αναφέρεται ως GLint εκτός αυτού του εγγράφου και δεν είναι απαραίτητα ισοδύναμος με τον τύπο int του C. Μια εφαρμογή μπορεί να χρησιμοποιεί περισσότερα bits από τον αριθμό που υποδεικνύεται στον πίνακα για να αντιπροσωπεύει έναν τύπο GL. Ωστόσο, δεν απαιτείται σωστή ερμηνεία των ακέραιων τιμών εκτός του ελάχιστου εύρους.



Εικόνα 3.1. Δομικό διάγραμμα της GL**.**

3.4 ΣΥΝΟΛΟ ΧΑΡΑΚΤΗΡΩΝ

(John Kessenich, 7 February 2013)Το σύνολο χαρακτήρων πηγής που χρησιμοποιείται για τις γλώσσες σκίασης OpenGL, εκτός από τα σχόλια, είναι ένα υποσύνολο του UTF-8. Περιλαμβάνει τους ακόλουθους χαρακτήρες:

- Τα γράμματα a-z, A-Z και η υπογράμμιση (_).
- Τους αριθμούς **0-9**.
- Τα σύμβολα περίοδος (.), συν (+), παύλα (-), κάθετος (/), αστερίσκος (*), τοις εκατό (%), γωνίες αγκύλες (<and>), αγκύλες([and]), παρενθέσεις((and)), άγκιστρα ({and}),caret (^), κάθετες ράβδους (|), ampersand (&), tilde (~), ίσον(=), θαυμαστικό(!), άνω κάτω τελεία(:), άνω τελεία (;), κόμμα (,) και ερωτηματικό (?).
- Το σύμβολο αριθμού (#) για τη χρήση του προεπεξεργαστή.
- Η αντίστροφη κάθετος (\) ως χαρακτήρας συνέχισης γραμμής όταν χρησιμοποιείται ως τελευταίος χαρακτήρας μιας γραμμής, ακριβώς πριν από μια νέα γραμμή.
- Λευκός χώρος: ο χαρακτήρας χώρου, η οριζόντια καρτέλα, η κατακόρυφη καρτέλα, η τροφοδοσία φόρμας, η επιστροφή μεταφοράς και η γραμμή τροφοδοσίας.

Ένα σφάλμα χρόνου μεταγλώττισης θα δοθεί αν κάποιος άλλος χαρακτήρας χρησιμοποιηθεί εκτός ενός σχολίου.

Δεν υπάρχουν διφθογγοι ή τρίφθογγοι. Δεν υπάρχουν ακολουθίες διαφυγής ή άλλες χρήσεις της αντίστροφης κάθετης προς τη χρήση ως χαρακτήρας συνέχειας γραμμής.

Οι γραμμές είναι σχετικές με τα διαγνωστικά μηνύματα του μεταγλωττιστή και τον προεπεξεργαστή. Τερματίζονται με επιστροφή μεταφοράς ή με τροφοδοσία γραμμής. Εάν και οι δύο χρησιμοποιούνται μαζί, θα θεωρηθεί ως μόνο ένας τερματισμός μιας γραμμής. Για το υπόλοιπο του παρόντος εγγράφου, οποιοσδήποτε από αυτούς τους συνδυασμούς αναφέρεται απλά ως νέα γραμμή.

Γενικά, η χρήση αυτού του συνόλου χαρακτήρων από τη γλώσσα είναι ευαίσθητη στη διάκριση πεζών-κεφαλαίων

Δεν υπάρχουν τύποι δεδομένων χαρακτήρων ή συμβολοσειρών, επομένως δεν συμπεριλαμβάνονται χαρακτήρες που περιέχουν τιμές.

Δεν υπάρχει χαρακτήρας στο τέλος του αρχείου.

3.5 SOURCE STRINGS (ΣΥΜΒΟΛΟΣΕΙΡΕΣ ΠΗΓΗΣ)

Η πηγή για ένα μόνο shader είναι μια σειρά από συμβολοσειρές χαρακτήρων από το σύνολο χαρακτήρων. Ένας απλός shader γίνεται από τη σύζευξη αυτών των συμβολοσειρών. Κάθε συμβολοσειρά μπορεί να περιέχει πολλές γραμμές, χωρισμένες με νέες γραμμές. Δεν χρειάζεται να υπάρχουν νέες γραμμές σε μια συμβολοσειρά. Μπορεί να σχηματιστεί μία γραμμή από πολλαπλά strings. Δεν εισάγονται νέες γραμμές ή άλλοι χαρακτήρες από την εκτέλεση όταν αυτή συνενώνει τις συμβολοσειρές για να σχηματίσουν ένα μόνο shader. Πολλαπλοί shaders μπορούν να συνδεθούν μεταξύ τους για να σχηματίσουν ένα ενιαίο πρόγραμμα.

Τα διαγνωστικά μηνύματα που επιστρέφονται από τη σύνταξη ενός shader πρέπει να προσδιορίζουν τόσο τον αριθμό γραμμής μέσα σε μια συμβολοσειρά όσο και τη σειρά συμβολοσειρών προέλευσης στην οποία αναφέρεται το μήνυμα. Οι συμβολοσειρές πηγής υπολογίζονται διαδοχικά με την πρώτη συμβολοσειρά να είναι η σειρά 0. Οι αριθμοί γραμμών είναι ένας αριθμός μεγαλύτερος από τον αριθμό των νέων γραμμών που έχουν υποβληθεί σε επεξεργασία, συμπεριλαμβανομένης της καταμέτρησης των νέων γραμμών που θα αφαιρεθούν από τον χαρακτήρα γραμμής συνέχισης (\).

Οι γραμμές που χωρίζονται από τον χαρακτήρα συνέχειας γραμμής που προηγείται μιας νέας γραμμής συνενώνονται μεταξύ τους πριν από την επεξεργασία ή την προεπεξεργασία των σχολίων. Δεν υπάρχει λευκός χώρος για τον χαρακτήρα συνέχειας γραμμής. Δηλαδή, ένα ενιαίο token(keyword, identifier, integer-constant, floating constant, operator) θα μπορούσε να σχηματιστεί με τη σύζευξη λαμβάνοντας τους χαρακτήρες στο τέλος μιας γραμμής που τους συνέδεσε με τους χαρακτήρες στην αρχή της επόμενης γραμμής.

```
float f\
oo;
// forms a single line equivalent to “float foo;”
// (assuming '\' is the last character before the new line and “oo”
are
```

```
// the first two characters of the next line)
```

3.6 ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ

Ακολουθούν οι λέξεις-κλειδιά της γλώσσας και (μετά από προεπεξεργασία) μπορούν να χρησιμοποιηθούν μόνο όπως περιγράφεται σε αυτήν την προδιαγραφή αλλιώς προκύπτουν σφάλματα χρόνου συλλογής:

attribute const uniform varying buffer shared
coherent volatile restrict readonly writeonly
atomic_uint
layout
centroid flat smooth noperspective
patch sample
break continue do for while switch case default
if else
subroutine
in out inout
float double int void bool true false
invariant precise
discard return
mat2 mat3 mat4 dmat2 dmat3 dmat4
mat2x2 mat2x3 mat2x4 dmat2x2 dmat2x3 dmat2x4
mat3x2 mat3x3 mat3x4 dmat3x2 dmat3x3 dmat3x4
mat4x2 mat4x3 mat4x4 dmat4x2 dmat4x3 dmat4x4
vec2 vec3 vec4 ivec2 ivec3 ivec4 bvec2 bvec3 bvec4 dvec2 dvec3 dvec4
uint uvec2 uvec3 uvec4
lowp mediump highp precision
sampler1D sampler2D sampler3D samplerCube
sampler1DShadow sampler2DShadow samplerCubeShadow
sampler1DArray sampler2DArray
sampler1DArrayShadow sampler2DArrayShadow
isampler1D isampler2D isampler3D isamplerCube
isampler1DArray isampler2DArray
usampler1D usampler2D usampler3D usamplerCube
usampler1DArray usampler2DArray
sampler2DRect sampler2DRectShadow isampler2DRect usampler2DRect
samplerBuffer isamplerBuffer usamplerBuffer
sampler2DMS isampler2DMS usampler2DMS
sampler2DMSArray isampler2DMSArray usampler2DMSArray
samplerCubeArray samplerCubeArrayShadow isamplerCubeArray usamplerCubeArray
image1D iimage1D uimage1D
image2D iimage2D uimage2D
image3D iimage3D uimage3D
image2DRect iimage2DRect uimage2DRect
imageCube iimageCube uimageCube
imageBuffer iimageBuffer uimageBuffer

image1DArray iimage1DArray uimage1DArray
image2DArray iimage2DArray uimage2DArray
imageCubeArray iimageCubeArray uimageCubeArray
image2DMS iimage2DMS uimage2DMS
image2DMSArray iimage2DMSArray uimage2DMSArray
struct

Επιπλέον, όλα τα αναγνωριστικά(identifiers) που περιέχουν δύο διαδοχικές υπογραμμίσεις (__) διατηρούνται ως πιθανές μελλοντικές λέξεις-κλειδιά.

Η περαιτέρω ανάλυση της χρήσης της Open GL δεν κρίνεται σκόπιμη δεδομένου ότι ο αναγνώστης της παρούσας εργασίας καλείται να γνωρίζει κάποια βασικά στοιχεία προκειμένου να κατανοήσει τον κεντρικό κορμό του θέματος.

4 ΤΑ ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ ΤΗΣ GLSL

(Rodriguez)Η Γλώσσα Σκίασης OpenGL βασίζεται στο ANSI C. Πολλά χαρακτηριστικά της γλώσσας προγραμματισμού C έχουν ενσωματωθεί στη GLSL, ενώ αυτά που αντιβαίνουν στην εκτέλεση ή τη γλωσσική απλότητα έχουν αφαιρεθεί.

Όλοι οι τύποι shader GLSL (vertex, fragment, geometry και compute) χρησιμοποιούν την ίδια γλώσσα. Εδώ θα μάθουμε τα βασικά της γλώσσας και τα κοινά στοιχεία μεταξύ κάθε τύπου shader. Συγκεκριμένα, θα μιλήσουμε για τα ακόλουθα θέματα:

- Βασικά στοιχεία γλώσσας
- Μεταβλητές εισόδου / εξόδου shader

Επειδή η GLSL είναι πολύ κοντά στη γλώσσα C, δεν θα εκθέσουμε μια πλήρη και εξαντλητική λίστα με κάθε στοιχείο γλώσσας. Θα επικεντρωθούμε μόνο στις διαφορές μεταξύ GLSL και C, και η μεγαλύτερη διαφορά από όλες αυτές είναι ότι η GLSL δεν έχει δείκτες.

4.1 Η ΓΛΩΣΣΑ

Όταν ξεκινάμε την κωδικοποίηση ενός shader, πρέπει να έχουμε κατά νου ένα σημαντικό πράγμα: ποια έκδοση GLSL πρόκειται να κωδικοποιήσουμε. Η έκδοση GLSL είναι πάντα συνδεδεμένη με μια συγκεκριμένη έκδοση της OpenGL, έτσι ώστε να επιλέξουμε μια έκδοση GLSL που να υποστηρίζει τα χαρακτηριστικά που χρειαζόμαστε, είμαστε επίσης συνδεδεμένοι με την ελάχιστη έκδοση OpenGL που υποστηρίζει αυτή την έκδοση GLSL.

Καθώς θα μιλήσουμε για compute shaders, πρέπει να πάμε στην ελάχιστη έκδοση που τους υποστηρίζει με φυσικό τρόπο (όχι μέσω επεκτάσεων), και αυτή είναι η GLSL Version 4.30.6.

4.2 ΒΑΣΙΚΑ ΣΤΟΙΧΕΙΑ ΓΛΩΣΣΑΣ

Πριν ξεκινήσουμε, απαιτείται μια βασική κατανόηση και επάρκεια στη C. Η OpenGL είναι διαθέσιμη σε διάφορες γλώσσες προγραμματισμού όπως Java, Python ή C #. Ωστόσο, θα επικεντρωθούμε στις έννοιες C / GLSL.

4.3 ΟΔΗΓΙΕΣ

Οι οδηγίες πάντα καταλήγουν σε ένα ερωτηματικό και μπορεί να υπάρχουν περισσότερες από μία ανά γραμμή:

```
   c = cross(a, b);
vec4 g; g = vec4(1, 0, 1, 1);
```

Ένα μπλοκ οδηγιών δημιουργείται τοποθετώντας τα μέσα σε άγκιστρα. Όλες οι μεταβλητές που δηλώνονται μέσα σε ένα μπλοκ θα καταστραφούν όταν τελειώσει το μπλοκ. Εάν δύο μεταβλητές έχουν το ίδιο όνομα - ένα δηλωμένο έξω από το μπλοκ (επίσης αποκαλούμενο πεδίο) και ένα άλλο δηλωμένο στο μπλοκ από προεπιλογή, η εσωτερική μεταβλητή είναι αυτή που θα αναφέρεται:

```
   float a = 1.0;
   float b = 2.0;
   {
   float a = 4.0;
   float c = a + 1.0; // c -> 4.0 + 1.0
   }
b = b + c; // wrong statement. Variable c does not exist here
```

Οι συστοιχίες ή τα κενά διαστήματα δεν αλλάζουν τη σημασιολογία της γλώσσας. Μπορούμε να τα χρησιμοποιήσουμε για να διαμορφώσουμε τον κώδικα όπως επιθυμούμε.

4.4 ΒΑΣΙΚΟΙ ΤΥΠΟΙ

Η GLSL είναι πολύ πλούσια σε σχέση με τους βασικούς της τύπους. Εκτός από τους δεδομένους τύπους C, έχουν προστεθεί και κάποιοι άλλοι - κυρίως για να αντιπροσωπεύουν διανύσματα ή για να εκθέσουν την εσωτερική αρχιτεκτονική GPU.

Ακολουθεί η πλήρης λίστα βασικών τύπων:

•`Bool`: Αυτό μπορεί να έχει μόνο δύο πιθανές τιμές - true ή false

•`Int`: Οι δύο τύποι int είναι οι εξής:

- `Int` (κανονική ακέραια τιμή)
- `Uint` (ακέραια τιμή χωρίς υπογραφή)

•`Sampler` (τύποι που αντιπροσωπεύουν υφές):

- `sampler1D, sampler2D, sampler3D`

•`float`

•**Διανύσματα**:

- `bvec2, bvec3, bvec4` (διανύσματα από 2, 3 και 4 στοιχεία Boolean)
- `ivec2, ivec3, ivec3` (διανύσματα από 2, 3 και 4 ακέραια στοιχεία)
- `uvec2, uvec3, uvec4` (διανύσματα από 2, 3 και 4 μη υπογεγραμμένους ακέραιους)
- `vec2, vec3, vec4` (διανύσματα από 2, 3, και 4 floats, με ενιαία ακρίβεια)

° `dvec2, dvec3, dvec4` (διανύσματα από 2, 3 και 4 floats, και διπλή ακρίβεια)

• **Πίνακες**: Οι πίνακες αποτελούνται πάντα από αριθμούς κινητής υποδιαστολής (το πρόθεμα d σημαίνει διπλή ακρίβεια):

° `mat2, mat3, mat4` (πίνακες 2 x 2, 3 x 3 και 4 x 4)

° `dmat2, dmat3, dmat4` (πίνακες 2 x 2, 3 x 3 και 4 x 4)

° `mat2x3, mat2x4, mat3x2, mat3x4, mat4x2, mat4x3` (ο πρώτος αριθμός αναφέρεται σε στήλες και ο δεύτερος σε σειρές)

° `dmat2x3, dmat2x4, dmat3x2, dmat3x4, dmat4x2, dmat4x3` (Ο πρώτος αριθμός αναφέρεται σε στήλες και ο δεύτερος σε σειρές)

4.5 ΜΕΤΑΒΛΗΤΟΙ ΑΡΧΙΚΟΠΟΙΗΤΕΣ

Για να αρχικοποιήσουμε ή να εκχωρήσουμε μια τιμή σε μια μεταβλητή, πρέπει να χρησιμοποιήσουμε τον κατασκευαστή του τύπου στη μεταβλητή. Ας δούμε μερικά παραδείγματα:

```
   float a = 1.0;
   bool switch = false; // Ok
   ivec3 a = ivec3(1, 2, 3); // Ok
   uvec2 a = uvec2(-1, 2); // Error, uivec2 is unsigned
   vec3 a(1.0, 2.0); // Error, you must assign the constructor
vec3 a = vec3(1.0, 2.0); // Ok
```

Ένα πολύ χρήσιμο τέχνασμα για την αρχικοποίηση του διανύσματος είναι ότι υπάρχουν πολλοί κατασκευαστές που είναι διαθέσιμοι. Για παράδειγμα, μπορούμε να κατασκευάσουμε `vec4` από `vec3` και `float`, ή δύο `vec2`, ή `float` και `vec3`. Όλοι οι πιθανοί συνδυασμοί στοιχείων που τελικά ταιριάζουν σε μέγεθος με τον κατασκευαστή στόχου είναι διαθέσιμοι:

```
   vec4 a = vec4(1.0, vec3(0.0, 1.0, 0.0));
   vec4 a = vec4(vec3(0.0, 1.0, 0.0), 0.9);
vec4 a = vec4(vec2(1.0, 1.0), vec2(0.5, 0.5));
```

Ένα διάνυσμα μπορεί να θεωρηθεί ως δομές ή συστοιχίες. Τα πεδία της δομής ενός διανύσματος που είναι προκαθορισμένα από τη γλώσσα και τα μεγέθη των συστοιχιών είναι τα αναμενόμενα μόνο αν κοιτάξουμε το όνομα του τύπου (το μέγεθος του vec3 είναι 3).

Για τα διανύσματα, τα ακόλουθα είναι τα έγκυρα ονόματα για τα πεδία της δομής (οι τρεις ομάδες είναι συνώνυμες):

• {x, y, z, w} χρήσιμο κατά την πρόσβαση σε διανύσματα που αντιπροσωπεύουν θέσεις
• {r, g, b, a} χρήσιμο κατά την πρόσβαση σε διανύσματα που αντιπροσωπεύουν χρώματα
• {s, t, p, q} χρήσιμο κατά την πρόσβαση σε διανύσματα που αντιπροσωπεύουν

συντεταγμένες υφής

Επίσης, μπορούμε να κάνουμε ευρετηρίο των διανυσμάτων χρησιμοποιώντας τους δείκτες:

```
vec2 p;
p[0] = 1.0; // ok
p.x = 1.0; // Ok
p.y = 2.0; // Ok
p.z = 3.0; // Illegal, p is a vec2, only has 2 elements
```

Η GLSL μας επιτρέπει να κάνουμε swizzle τα στοιχεία ενός διανύσματος (δηλαδή, να κατασκευάσουμε ένα νέο διάνυσμα διπλασιάζοντας ή αναδιατάσσοντας τα στοιχεία του πρώτου). Για να γίνει αυτό δυνατό, η GLSL επιτρέπει πράγματα τόσο χρήσιμα όπως τα επόμενα:

```
vec4 color1 = vec4(0.5, 0.2, 1.0, 1.0); // RGBA color;
// Let's convert color to abgr
vec4 color2 = color1.abgr; // equivalent to color1.wzyx
// Let's make a grey color based only on the red component
vec4 redGray = color1.rrrr;
float red = color1.r;
// Let's swizzle randomly but in valid way
vec4 color3 = color1.gbgb;
Vec4 color4 = vec4(color1.rr, color2.bb); // .rr .bb are vec2
Vec4 color5 = color1.tptp; // the same than .gbgb
vec4 color6 = color1.yzyz; // the same than .gbgb and .tptp
color6.xy = vec2(1.0, 0.0);
color6[3] = 2.0;
// Some invalid swizzles
vec2 p;
p = color1.rgb; // .rgb is vec3
p.xyz = color.rgb; // .xyz doesn't match p's size
p[2] = 3.0; // index out of bounds.
vec4 color7 = color1.xxqq; // Ilegal, components don't come from the
same set
```

Μιλώντας για πίνακες, ακολουθούν τους ίδιους κανόνες σε σύγκριση με τα διανύσματα, εκτός από το γεγονός ότι ένας πίνακας είναι μια συστοιχία διανυσμάτων στήλης και οι πίνακες GLSL είναι πίνακες στήλης. Αυτό σημαίνει ότι ο πρώτος δείκτης είναι ο δείκτης στήλης και ο δεύτερος δείκτης είναι ο δείκτης γραμμής:

```
mat4 m;
m[0] = vec4(1.0); // put the first column to 1.0 in each element
m[1][1] = 2.0; // put the second diagonal element to 2.0
m[2] = color3.xxyy; // Insert a swizzled vector into the third
column
```

4.6 ΔΡΑΣΤΗΡΙΟΤΗΤΕΣ ΔΙΑΝΥΣΜΑΤΟΣ ΚΑΙ ΠΙΝΑΚΑ

Από προεπιλογή, οι πράξεις διανύσματος και πίνακα και ορισμένοι αριθμητικοί χειριστές έχουν υπερφορτωθεί για να ταιριάζουν με γραμμικές αλγεβρικές συμβάσεις. Σχεδόν όλες οι λειτουργίες είναι συνιστώσες, αλλά οι πολλαπλασιασμοί πίνακα με πίνακα και πίνακα με

διάνυσμα ακολουθούν τους κανόνες των γραμμικών αλγεβρικών μετασχηματισμών:

```
   mat3 R, T, M;
   vec3 v, b;
   float f;
   // Initialize f, b, v, R and T with some values
   // …
   // Component wise example
   b = v + f;
   /* this stands for:
   b.x = v.x + f;
   b.y = v.y + f;
   b.z = v.z + f;
   The same is applied with the product operator, even between two
   vector types */
   // Linear algebra transform
   // Vectors are considered column vectors
   b = T * v;
   /* Is equivalent to:
   b.x = T[0].x * v.x + T[1].x * v.y + T[2].x * v.z;
   b.y = T[0].y * v.x + T[1].y * v.y + T[2].y * v.z;
   b.z = T[0].z * v.x + T[1].z * v.y + T[2].z * v.z; */
   M = T * R;
/* Too long for putting that operation here, but it follows the
rules of column matrices*/
```

Συνοψίζοντας και μιλώντας για πίνακες μετασχηματισμού, αν θέλουμε να μετασχηματίσουμε ένα διάνυσμα με ένα πίνακα μετασχηματισμού, απλά το πολλαπλασιάζουμε. Αν θέλουμε να συνενώσουμε μετασχηματισμούς, απλά τους γράφουμε φυσικά από δεξιά προς τα αριστερά:

```
   b = T * v; // Translate v with the translation matrix T
   b = R * T * v; // Translate and then rotate vector v
   vertex = projection * modelview * vertex_position;

/* Transform a vertex position from model coordinates to clip
coordinate system. This is the way we always will go from model
space to clip space.*/
```

4.7 CASTINGS ΚΑΙ ΜΕΤΑΤΡΟΠΕΣ

Μπορούμε να “ρίξουμε” τύπους μόνο σε άλλους τύπους, όταν δεν πρόκειται να υπάρξουν προβλήματα ακρίβειας στη διαδικασία, αλλά άλλοι τύποι μετασχηματισμών πρέπει να γίνουν με σαφήνεια. Για παράδειγμα, μπορούμε να “πετάξουμε” σιωπηρά από `int` σε `uint`, από `int` σε `float`, από `float` σε `double`, και ούτω καθεξής.
Διαφορετικά, πρέπει να εκτελέσουμε ρητά τα cast ή θα καταλήξουμε σε σφάλματα σύνταξης. Το ρητό casting γίνεται μέσω κατασκευαστή. Για παράδειγμα:

```
   float threshold = 0.5;
   int a = int(threshold); // decimal part is dropped, so a = 0;
   double value = 0.334lf
   float value2 = float(value);
   bool c = false;
   value2 = float(c); // value2 = 0.0;
   c = true;
value2 = float(c); // value2 = 1.0;
```

Σε γενικές γραμμές, εάν έχουμε προβλήματα με σιωπηρά castings, θα πρέπει να μεταβούμε στον ρητό τρόπο (ή καλύτερα, να πραγματοποιούμε πάντα ρητές μετατροπές).

4.8 ΣΧΟΛΙΑ ΚΩΔΙΚΑ

Τα σχόλια κώδικα είναι χρήσιμα για τη δημιουργία παρατηρήσεων στον κώδικα για να διευκρινιστούν ορισμένες λειτουργίες για περαιτέρω ανάγνωση. Δεν είναι ασυνήθιστο να βρούμε την κωδικοποίηση λίγο πολύ κουραστική. Κάτω από τέτοιες συνθήκες, είναι πάντα καλύτερο να συμπεριληφθούν τα σχόλια για αναφορές αργότερα. Υπάρχουν δύο μορφές σχολίων κώδικα: σχόλια μιας γραμμής ή μπλοκ γραμμών σχόλιων:

```
   // This is a single-line comment. This line will be ignored by the
   compiler
   /* This is a block of lines comment. It's initialized
   with a slash and an asterisk and ends with an asterisk
   and slash. Inside this block you can put anything, but
be careful because these kind of comments can't be nested */
```

4.9 ΕΛΕΓΧΟΣ ΡΟΗΣ

Όπως ακριβώς με όλες σχεδόν τις άλλες γλώσσες προγραμματισμού, μπορούμε να ελέγξουμε τη ροή του κώδικα μας, ελέγχοντας τις συνθήκες Boolean:

```
   if(a > threshold)
ligthColor = vec4(1.0, 1.0, 1.0, 1.0); //single new line, no matter
the indentation
```

Εναλλακτικά, τοποθετούμε την επόμενη πρόταση κώδικα στην ίδια γραμμή:

```
if(a > threshold) lightColor = vec4(1.0, 1.0, 1.0, 1.0);
```

Μπορούμε επίσης να χρησιμοποιήσουμε την εντολή `else` για να εκτελέσουμε ενέργειες όταν η συνθήκη `if` δεν είναι αληθής:

```
   if(a > threshold)
   lightColor = vec4(1.0, 1.0, 1.0, 1.0);
   else
lightColor = vec4(1.0, 0.0, 0.0, 1.0); // when a <= threshold
```

Η τελευταία παραλλαγή αφορά όταν θέλουμε να δοκιμάσουμε κάποιες συνθήκες χρησιμοποιώντας το ίδιο εύρος, για παράδειγμα, όταν χρησιμοποιούμε μία ή περισσότερες δηλώσεις `if`:

```
   if(a < threshold)
   lightColor = vec4(1.0, 0.0, 0.0, 1.0);
   else if(a == threshold)
   lightColor = vec4(0.0, 1.0, 0.0, 1.0);
   else if((a > threshold + 1) && (a <= threshold + 2))
   lightColor = vec4(0.0, 0.0, 1.0, 1.0);
   else
lightColor = vec4(1.0, 1.0, 1.0, 1.0);
```

Οι γλώσσες που μοιάζουν με το C δεν βασίζονται σε εσοχές, όπως για παράδειγμα η Python. Εάν θέλουμε να ορίσουμε περισσότερες από μία εντολές, πρέπει να χρησιμοποιήσουμε αποκλειστικά τις ομάδες:

```
   if(a > threshold)
   {
   lightColor = vec4(1.0, 1.0, 1.0, 1.0);
   baseColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Φυσικά, μπορούμε να αποθηκεύουμε όρους όπως σε οποιαδήποτε άλλη γλώσσα προγραμματισμού. Στη GLSL, είναι απολύτως σωστό και νόμιμο (αλλά κακό στην εκτέλεση, δεδομένου ότι πάντα είναι υπό όρους):

```
   if(a > threshold1)
   {
   if(b > threshold2)
   {
   LightColor = vec4(1.0, 0.0, 0.0, 1.0);
   }
   lightColor = vec4(0.0, 1.0, 0.0, 1.0);
   }
   else
lightColor = vec4(0.0, 0.0, 0.0, 1.0);
```

Όταν πρέπει να ελέγξουμε διαφορετικές δυνατότητες της τιμής μιας μοναδικής μεταβλητής, μπορούμε να χρησιμοποιήσουμε τη δήλωση `switch-case`:

```
   switch(myVariable)
   {
   case 1: // when myVariable's value is 1
   lightColor = vec4(1.0, 0.0, 0.0, 1.0);
   break;
   case 2: // when myVariable's value is 2
   lightColor = vec4(0.0, 1.0, 0.0, 1.0);
   break;
```

```
   case 3: // when myVariable's value is 3
   lightColor = vec4(0.0, 0.0, 1.0, 1.0);
   break;
   default: // in any other case.
   lightColor = vec4(1.0, 1.0, 1.0, 1.0);
}
```

Υπάρχουν μερικά πράγματα που πρέπει να εξηγήσουμε εδώ. Πρώτον, πρέπει να έχουμε παρατηρήσει τη δήλωση `break`. Πρέπει να την τοποθετήσουμε εκεί για να διαχωρίσουμε τις περιπτώσεις. Εάν δεν το κάνουμε, και υποθέσουμε ότι η `case 2` δεν περιέχει δήλωση `break`, η εκτέλεση κώδικα θα συνεχιστεί στην `case 3`.

Δεν πρέπει να ξεχνάμε να βάζουμε `breaks` όπου χρειάζεται.

Έχουμε επίσης θέσει τη ρήτρα `default` εκεί. Αυτή η ρήτρα είναι που θα βρίσκεται ο κώδικας όταν όλες οι άλλες περιπτώσεις αποτύχουν. Σε αυτό το παράδειγμα, αν η τιμή του `myVariable` είναι 4, 5, ή 42342344, ο κώδικας θα εισαχθεί στη ρήτρα `default`. Αυτή η ρήτρα είναι προαιρετική, επομένως δεν χρειάζεται να την τοποθετούμε εκεί εάν δεν την χρειαζόμαστε.

Μια εντολή `switch-case` είναι ισοδύναμη με μια αλυσίδα δηλώσεων `if-else if` . Αλλά αν μπορούμε, χρησιμοποιούμε τη δήλωση `switch` αντί `if-else`.

```
switch statement
switch(a)
{
case 1:
offset = offset + 1.3;
offset2 = offset;
break;
case 2:
offset = offset + 3.4;
break;
default:
offset = 0;
}
```

```
if-else statement
if(a == 1)
{
offset = offset + 1.3;
offset2 = offset;
}
else if(a == 2)
offset = offset + 3.4;
else
offset = 0;
```

Μπορούμε επιπροσθέτως να αναφερθούμε σε loops,arrays,μεταβλητές εισόδου και εξόδου, και πλήθος άλλων λειτουργιών που αφορούν στο χειρισμό και την κατανόηση της GLSL, ώστοσο κρίνεται σκόπιμο να περιοριστούμε στις αναφορές μας σε αυτό το κομμάτι, εφόσον δεν αφορά στο βασικό τμήμα της στόχευσης μας.

5 VERTEX SHADERS

Οι vertex shaders είναι υπεύθυνοι για το μετασχηματισμό της εισερχόμενης γεωμετρίας σε κάτι κατάλληλο για να ραστεροποιηθεί, σύμφωνα με τους νόμους του αγωγού απόδοσης(Rendering Pipeline). Για να το κάνουμε να δουλέψει, οι είσοδοι(inputs) και οι έξοδοι(outputs) του shader πρέπει να είναι πολύ καλά καθορισμένες.

Σε αυτό το κεφάλαιο θα δούμε πώς πρέπει να προετοιμαστούν οι είσοδοι και πώς μπορούμε να υπολογίσουμε τις εξόδους. Επίσης, θα μιλήσουμε εκτενώς για τις πράξεις που επιτρέπεται να εκτελέσουμε.

Ένας vertex shader εκτελείται μία φορά και μόνο μία φορά για κάθε κορυφή που στέλνεται στη GPU(μονάδα επεξεργασίας γραφικών). Μέσα σε έναν vertex shader , έχουμε πρόσβαση σε όλες τις πληροφορίες σχετικά με αυτήν την κορυφή, αλλά δεν μπορούμε να έχουμε πρόσβαση στις άλλες “συγγενείς” κορυφές του πρωτοκόλλου που βρίσκεται υπό επεξεργασία.

Δεν έχει σημασία για τον vertex shader ποιος τύπος πρωτοκόλλου και πώς τον ορίσαμε πριν το στείλουμε στη GPU (με ευρετήριο(indexed),χωρίς ευρετήριο(non-indexed), παρεμβαλλόμενο(interleaved), μη παρεμβαλλόμενο(non-interleaved), VBO, VAO κ.ο.κ.).

Έτσι, στο τέλος, ένας vertex shader είναι μια μηχανή "δώσε μου μια κορυφή που θα μετασχηματίσω για σένα" και τίποτα άλλο. Τα πράγματα έχουν διατηρηθεί απλά όπως μπορούμε να δουμε.

Θα ολοκληρώσουμε αυτό το κεφάλαιο με ένα παράδειγμα όπου θα μιλήσουμε για πιθανά προβλήματα, λύσεις και τεχνικές εντοπισμού σφαλμάτων όταν δεν έχουμε τα αναμενόμενα αποτελέσματα.

5.1 ΕΙΣΟΔΟΙ VERTEX SHADER (INPUTS)

Ένας vertex shader μπορεί να έχει μόνο δύο διαφορετικά είδη εισόδων(inputs): χαρακτηριστικά κορυφής και ομοιόμορφες μεταβλητές.

5.1.1 ΧΑΡΑΚΤΗΡΙΣΤΙΚΑ ΚΟΡΥΦΗΣ (VERTEX)

Ένα χαρακτηριστικό κορυφής(vertex) είναι απλώς οι πληροφορίες που περάσαμε στον shader, σε μια βάση ανά κορυφή, κατά μήκος της απλής κορυφής θέσης στον κόσμο. Παραδείγματα χαρακτηριστικών κορυφής μπορεί να είναι:

• Συντεταγμένες υφής
• Μέσοι όροι(normals)
• Εφαπτομένες
• Χρώματα ανά κορυφή

Λόγω της εξελισσόμενης φύσης της OpenGL, οι λεπτομέρειες τείνουν να είναι γενικευμένες. Οι συγγραφείς προδιαγραφών OpenGL προσπαθούν να ορίσουν τα δεδομένα όσο το δυνατόν πιο ομοιόμορφα. Στις πρώτες ημέρες προγραμματισμού του shader υπήρχαν

συγκεκριμένα χαρακτηριστικά με συγκεκριμένα ονόματα για συντεταγμένες υφής, normals, χρώματα κορυφών κλπ. Τώρα, όλα τα χαρακτηριστικά είναι γενικά και δεν έχουν ειδικά ονόματα. Είναι απλά buffer χαρακτηριστικών κορυφής.

Προκειμένου να δείξουμε πώς τα χαρακτηριστικά συνδέονται με τον vertex shader, πρέπει να δούμε πώς να τα ρυθμίσουμε στην εφαρμογή υποδοχής της OpenGL.

Πρώτον, πρέπει να δημιουργήσουμε και να γεμίσουμε μια συστοιχία(array) κορυφών ρυθμισμένη με κορυφές και συντεταγμένες υφής (για παράδειγμα). Υπάρχουν πολλοί τρόποι για να επιτευχθεί αυτό. Θα παρουσιάσουμε ένα παρεμβαλλόμενο αντικείμενο συστοιχίας κορυφής (VAO) που θα κρατάει μία πλήρη κορυφή στο buffer (θέση και συντεταγμένες υφής) ταυτόχρονα, έτσι ώστε το buffer θα μοιάζει σαν XYZWSTXYZWST, με XYZW ως συντεταγμένες θέσης κορυφής και ST ως συντεταγμένες υφής:

```
   // Create vertex array object (VAO)
   glGenVertexArrays(1, &vaoID);
   glBindVertexArray(vaoID);
   // Create vertex buffer object (VBO)
   glGenBuffers(1, &vboID);
   glBindBuffer(GL_ARRAY_BUFFER, vboID);
   // Fill buffer with data
   glBufferData(GL_ARRAY_BUFFER,        bufferElementsCount        *
   sizeof(GLfloat), bufferData, GL_STATIC_DRAW);
   // Tell OpenGL how to locate the first attribute (positions)
   inside the buffer
glEnableVertexAttribArray(0);
   glVertexAttribPointer(0,  4,  GL_FLOAT,  GL_FALSE,  (4  +  2)*
   sizeof(GLfloat), NULL);
   // Tell OpenGL how to locate the second attribute (texture
   coordinates) inside the buffer
   glEnableVertexAttribArray(1);
   glVertexAttribPointer(1,  2,  GL_FLOAT,  GL_FALSE,  (4  +  2)  *
   sizeof(float),(void*)(4 * sizeof(float)));
   // Deactivate VAO
   glBindBuffer(GL_ARRAY_BUFFER, 0);
glBindVertexArray(0);
```

Σε αυτό το απόσπασμα κώδικα, έχουμε δημιουργήσει ένα VAO με δύο χαρακτηριστικά κορυφής. Στο χαρακτηριστικό με index(ευρετήριο) 0, έχουμε τοποθετήσει τις θέσεις των κορυφών, και στη συστοιχία με το index(ευρετήριο) 1, τις συντεταγμένες υφής.

Έτσι, τώρα που έχουμε τα δεδομένα μας έτοιμα, ας επιστρέψουμε στο shader. Πρέπει να γνωρίζουμε πώς να αποκτήσουμε πρόσβαση στο buffer μας από τον vertex shader. Παρόλο που υπάρχουν κάποιες επιλογές γι 'αυτό, χρησιμοποιούμε πάντα την πιο ρητή (και απλούστερη) μία. Ανακαλώντας το γεγονός ότι βάλαμε θέσεις στην slot(υποδοχή) 0 και τις συντεταγμένες της υφής στην slot(υποδοχή) 1, θα χρησιμοποιήσουμε τους αριθμούς slot για να αποκτήσουμε πρόσβαση σε δεδομένα από τον shader με αυτές τις πληροφορίες. Ένας κενός vertex shader θα μοιάζει με τον ακόλουθο:

```
   #version 430
```

```
   #pragma debug(on)
   #pragma optimize(off)
   // 0 -> the slot I set with the glEnableVertexAttribArray(0) call
   for the vertices' positions
   layout (location = 0) in vec4 position;
   // 1 -> the slot I set with the glEnableVertexAttribArray(1) call
   for the vertices' positions
   layout (location = 1) in vec2 texCoords;
   void main()
   {
   // Do something here
}
```

Όπως φαίνεται, έχουμε χρησιμοποιήσει τη λέξη-κλειδί `in`. Επίσης, λέμε ρητά στον shader σε ποια θέση έχουμε τοποθετήσει τα χαρακτηριστικά. Αυτός ο τρόπος καθορισμού χαρακτηριστικών θα μπορούσε να είναι καλός για το σύστημα rendering εάν χρησιμοποιούμε πάντα την ίδια διάταξη χαρακτηριστικών, αλλά εάν όχι, έχουμε την επιλογή να αφήσουμε την OpenGL να αντιστοιχίσει αυτόματα την slot και αργότερα να την ρωτήσουμε.

5.1.2 ΟΜΟΙΟΜΟΡΦΕΣ ΜΕΤΑΒΛΗΤΕΣ

Μπορούμε να περάσουμε τις μεμονωμένες μεταβλητές της εφαρμογής που δεν σχετίζονται με την σωλήνωση ή με οποιονδήποτε άλλο περιορισμό. Οι τιμές των μεταβλητών αυτών είναι σταθερές μέχρι να αλλάξουμε τις τιμές τους από την εφαρμογή υποδοχής της OpenGL.

Χρησιμοποιώντας αυτές τις μεταβλητές, μπορούμε να περάσουμε τους πίνακες μετασχηματισμού στον shader, τις παραμέτρους φωτισμού ή ό, τι μπορούμε να σκεφτούμε. Ας δημιουργήσουμε τον πρώτο λειτουργικό vertex shader.

Πρώτον, ας ανεβάσουμε τις τιμές των μεταβλητών στη GPU στην εφαρμογή υποδοχής της OpenGL:

```
   // Enable our shader program. Before doing any operation like
   setting uniforms or retrieving vertex attribute slot numbers, the
   shader must be activated/bound using this function
   glUseProgram(programID);
   // Value to be uploaded to the GPU
   GLfloat mdlv[16] = …; // Fill with proper values
   GLfloat proj[16] = …; // Fill with proper values
   // retrieve variable's slots from the shader using the variable's
   names in the shader
   Glint location = glGetUniformLocation(programID, "Modelview");
   // Upload the values to the shader
   glUniformMatrix2fv(location, 1, GL_FALSE, mdlv);
   location = glGetUniformLocation(programID, "Projection");
glUniformMatrix4fv(location, 1, GL_FALSE, proj);
```

Και αυτό είναι όλο. Οι τιμές είναι έτοιμες για χρήση στο shader. Ας ολοκληρώσουμε το

παράδειγμα GLSL:

```
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
layout (location = 0) in vec4 Position;
   // uniform matrices declaration. This must be done globally
   uniform mat4 Modelview;
   uniform mat4 Projection;
   void main()
   {
   // Let's compute the vertex position in clip coordinate system
   gl_Position = Projection * Modelview * Position;
}
```

Εκτός από το “ανέβασμα” μιας απλής συστοιχίας από float στοιχεία ως ομοιόμορφες τιμές, έχουμε δηλώσει μεταβλητές του τύπου `mat4`. Η GLSL θα φροντίσει αυτόματα να διαμορφώσει αυτές τις συστοιχίες στον τύπο που στοχεύουμε. Βέβαια, τα δεδομένα πρέπει να ταιριάζουν φυσικά. Δεν μπορούμε απλά να σχεδιάσουμε `bool` στο `mat4`, ή ακόμα και `float [9]` στο `mat4`. Πρέπει να είμαστε προσεκτικοί με αυτό.

Η `gl_Position` είναι μια ειδική (ενσωματωμένη) τιμή που πρέπει να κρατήσει τη μετασχηματισμένη θέση κορυφής. Αυτή είναι η μόνη απαίτηση για ένα vertex shader και είναι ο κύριος σκοπός του. Εάν δεν βάλουμε μια τιμή εκεί, θα καταστήσουμε το shader ακατάλληλο και κακοσχηματισμένο και θα πάρουμε ένα σφάλμα μεταγλωττιστή.

5.2 ΕΞΟΔΟΙ VERTEX SHADER (OUTPUTS)

Οι vertex shaders μπορούν να εξάγουν γενικές τιμές στο επόμενο στάδιο. Αυτές οι τιμές πρόκειται να περάσουν στη φάση geometry shading πρώτα, στη συνέχεια να ραστεροποιηθούν και τελικά να περάσουν στο στάδιο fragment shading σε μία από τις επιτρεπόμενες μορφές παρεμβολής. Μπορούμε να επιλέξουμε το είδος της παρεμβολής, αλλά θα χρησιμοποιήσουμε σε αυτή τη φάση μια γραμμική προοπτικά διορθωμένη παρεμβολή.

Για να καθορίσουμε μια μεταβλητή εξόδου, πρέπει να χρησιμοποιήσουμε τον τύπο παρεμβολής μαζί με τη λέξη κλειδί `out`:

```
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   layout (location = 0) in vec4 Position;
   layout (location = 1) in vec2 TexCoord;
   uniform mat4 Modelview;
   uniform mat4 Projection;
   // smooth = linearly perspective-correct interpolation
smooth out vec2 texCoordsInterpolated;
   void main()
   {
   gl_Position = Projection * Modelview * Position;
```

```
   // Write the vertex attribute into an output variable that will be
   interpolated in a later pipeline's stages
   texCoordsInterpolated = TexCoord;
}
```

5.3 ΣΧΕΔΙΑΖΟΝΤΑΣ ΕΝΑ ΑΠΛΟ ΔΕΙΓΜΑ ΓΕΩΜΕΤΡΙΑΣ

Προκειμένου να έχουμε ένα πρόγραμμα shader που να “τρέχει”, χρειαζόμαστε ένα vertex shader και ένα fragment shader. Μέχρι τώρα, δεν έχουμε μιλήσει για fragment shaders, αλλά θα χρειαστούμε ένα για να τρέξουμε τα δείγματα, οπότε θα παρουσιάσουμε εδώ ένα απλό fragment shader που θα χρησιμοποιηθεί σε συνδυασμό με τους επόμενους vertex shaders. Αυτό είναι το δείγμα του fragment shader, όσο το δυνατόν πιο απλό :

```
   // Fragment shader
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   uniform vec4 SolidColor;
   // Writing to this variable will set the current fragment's color
   out vec4 frameBufferColor;
   void main()
   {
   frameBufferColor = SolidColor;
}
```

Αυτός ο fragnent shader “βάφει” όλη την επιφάνεια του τριγώνου με ένα συμπαγές χρώμα, που παρέχεται μέσω μιας ομοιόμορφης μεταβλητής (οι fragment shaders έχουν επίσης ομοιόμορφες μεταβλητές).

Επιστρέφοντας στους vertex shaders , ας ξεκινήσουμε με έναν απλό. Το πιο απλό πράγμα που θα θέλαμε να κάνουμε είναι να δείξουμε τη γεωμετρία μας, όπως είναι, στην οθόνη, με ένα ενιαίο χρώμα. Επομένως, πρέπει να μετασχηματίσουμε τις κορυφές μας χρησιμοποιώντας τους αντίστοιχους πίνακες:

```
   // Vertex shader
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   layout (location = 0) in vec4 position;
   uniform mat4 Modelview;
   uniform mat4 Projection;
   void main()
   {
   gl_Position = Projection * Modelview * position;
}
```

Μπορούμε επίσης να υπολογίσουμε την προβολή modelview στην CPU και να περάσουμε το αποτέλεσμα στο shader. Μπορούμε να το κάνουμε να σταματήσει να εκτελείται πολλές φορές στον shader, επειδή ο πολλαπλασιασμός αυτός πάντα φέρνει το ίδιο

αποτέλεσμα (εφ 'όσον δεν αλλάζουν οι δύο πίνακες) για όλα τα πρωτόκολλα της τρέχουσας σχεδιαστικής κλήσης OpenGL. Έτσι, ο shader μας θα μοιάζει με τα εξής:

```
   // Vertex shader
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   layout (location = 0) in vec4 Position;
   uniform mat4 ProjectionModelView;
   void main()
   {
   gl_Position = ProjectionModelView * Position;
}
```

Εάν υπολογίσουμε τις θέσεις των κορυφών των απαιτούμενων τιμών για μια τσαγιέρα (ή οποιοδήποτε άλλο triangle mesh) και χρησιμοποιήσουμε το shader του τρέχοντος δείγματος, περνώντας ένα μισό κόκκινο χρώμα στην ομοιόμορφη μεταβλητή, θα έχουμε μια απόδοση όπως αυτή στο παρακάτω σχήμα:

5.4 ΠΑΡΑΜΟΡΦΩΣΗ ΔΕΙΓΜΑΤΟΣ ΓΕΩΜΕΤΡΙΑΣ

Τώρα που έχουμε κάνει σωστή απόδοση της γεωμετρίας μας, ας κάνουμε κάποιες στρεβλώσεις. Πρώτον, θα κλιμακώσουμε την τσαγιέρα στην οριζόντια κατεύθυνση. Θα χρειαστούμε έναν πίνακα κλίμακας(scale) και θα τον εφαρμόσουμε στη γεωμετρία ως το πρώτο μετασχηματισμό που θα εφαρμοστεί:

```
   // Vertex shader
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   layout (location = 0) in vec4 Position;
   uniform mat4 ProjectionModelView;
```

```
   void main()
   {
   // Remember that the first vec4 is the first column, not the first
   row, although in the code apparently it looks like a row because
   of the way it's written
   mat4 scale = mat4(vec4(1, 0, 0, 0),// scale in the X direction
   vec4(0, 1, 0, 0),// scale in the y direction
   vec4(0, 0, 1, 0),// scale in the z direction
   vec4(0, 0, 0, 1));
   gl_Position = ProjectionModelView * scale * Position;
}
```

Αυτός ο shader δεν έχει παραμόρφωση στον πίνακα κλίμακας(scale). Όλοι οι συντελεστές κλίμακας είναι 1,0, πράγμα που σημαίνει ότι όλα θα παραμείνουν στην αρχική τους κατάσταση. Αλλά ας δούμε τα επόμενα σχήματα όταν αλλάζουμε τους συντελεστές κλίμακας:

Για να παραμορφώσουμε την τσαγιέρα Α, χρησιμοποιήσαμε τον ακόλουθο πίνακα κλίμακας:

```
   mat4 scale = mat4(vec4(1, 0, 0, 0),
   vec4(0, 1.5, 0, 0), // 150% scale in Y direction
   vec4(0, 0, 1, 0),
   vec4(0, 0, 0, 1));
   For the (B) teapot, our matrix was:
   mat4 scale = mat4(vec4(1.5, 0, 0, 0), // 150% scale in X direction
   vec4(0, 1, 0, 0),
```

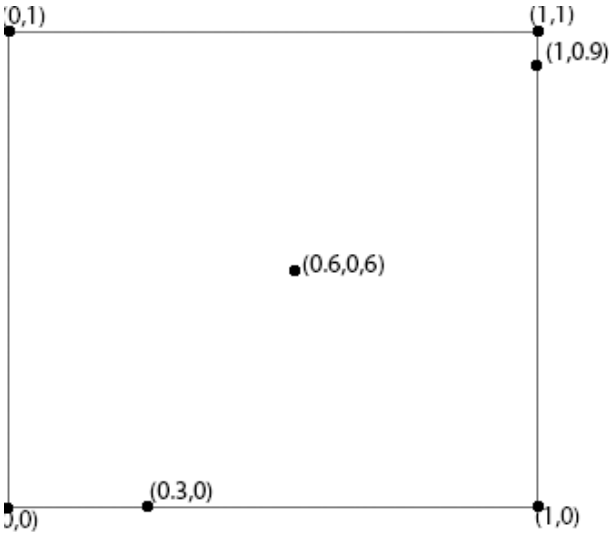
```
   vec4(0, 0, 1, 0),
   vec4(0, 0, 0, 1));
   And finally, the (C) teapot's scale matrix was:
   mat4 scale = mat4(vec4(1, 0, 0, 0),
   vec4(0, 1, 0, 0),
   vec4(0, 0, 3, 0), // 300% scale in Z direction
vec4(0, 0, 0, 1));
```

Όπως μπορούμε να δούμε, απλώς εφαρμόζοντας τον κατάλληλο πίνακα μετασχηματισμού, έχουμε την επιθυμητή παραμόρφωση. Φυσικά, υπάρχουν και άλλοι τύποι πινάκων μετασχηματισμού: οι μετατοπίσεις, οι περιστροφές και οι αλλαγές κλίμακας (και συνδυασμοί αυτών) είναι οι πιο ευρέως χρησιμοποιούμενες.

5.5 ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ ΠΑΡΕΜΒΟΛΕΙΣ

Άλλες τιμές εξόδου κορυφής εκτός από το `gl_Position` καλούνται συχνά παρεμβολείς. Αυτό συμβαίνει επειδή οι τιμές που αποθηκεύονται εκεί θα παρεμβληθούν στην επιφάνεια του πρωτοκόλλου στα μεταγενέστερα στάδια. Εν συντομία, ας υποθέσουμε ότι κάνουμε rendering μια σειρά. Στη συνέχεια, θέλουμε ένα διαφορετικό χρώμα vertex σε κάθε κορυφή, οπότε δημιουργούμε μια συστοιχία χαρακτηριστικών κορυφής που αποθηκεύει αυτά τα χρώματα. Τώρα υποθέτουμε ότι το χρώμα στην πρώτη κορυφή είναι καθαρό λευκό και το χρώμα στη δεύτερη κορυφή είναι καθαρό μαύρο. Αν βάλουμε τα χρώματα των κορυφών σε έναν παρεμβολέα και επιλέξουμε αυτήν την παρεμβαλλόμενη τιμή σε ένα fragment shader, κάθε fragment που πρόκειται να βαφεί θα παρουσιάσει ένα παρεμβαλλόμενο χρώμα ανάμεσα σε καθαρό μαύρο και καθαρό λευκό. Συνοψίζοντας, η γραμμή που σχεδιάζουμε θα σχηματίσει ένα gradient του γκρι, από καθαρό λευκό έως καθαρό μαύρο.

Χρησιμοποιώντας ένα σχήμα 2D όπως ένα τετράγωνο και τοποθετώντας σε κάθε γωνία τις τιμές (0,0), (1,0), (0,1) και (1,1) λαμβάνουμε το ακόλουθο διάγραμμα:

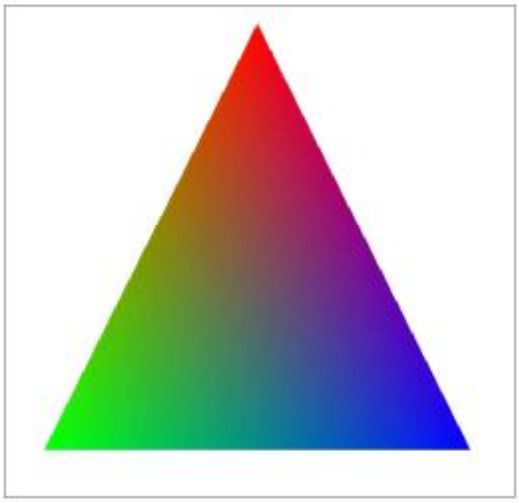


Όπως μπορούμε να δούμε, τα σημεία που δημιουργούνται ανάμεσα έχουν γραμμικώς παρεμβληθεί. Όσο πιο κοντά στη μια γωνία, τόσο πιο κοντά είναι οι τιμές.

Ο καλύτερος τρόπος να απεικονιστεί αυτό είναι με τα χρώματα. Ας φτιάξουμε ένα τρίγωνο με διαφορετικό χρώμα σε κάθε κορυφή: πράσινο στην κάτω αριστερή γωνία, κόκκινο στην πάνω γωνία, και μπλε στην κάτω δεξιά γωνία. Ο κώδικας για το ίδιο έχει ως εξής:

```
   // Vertex shader
   #version 430
   #pragma debug(on)
#pragma optimize(off)
   layout (location = 0) in vec4 Position;
   // Vertexcolor buffer mapping
   layout (location = 1) in vec4 VertexColor;
   // Interpolator for our vertex color
   smooth out vec4 interpolatedColor;
   uniform mat4 ProjectionModelView;
   void main()
   {
   interpolatedColor = VertexColor;
   gl_Position = ProjectionModelView * scale * Position;
   }
   // Fragment shader
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   // Here we are receiving the interpolated vertex color
   smooth in vec4 interpolatedColor;
   out vec4 frameBufferColor;
   void main()
   {
   frameBufferColor = interpolatedColor;
}
```

Αυτός ο shader θα παράγει την ακόλουθη απόδοση:



5.6 ΑΠΛΟΣ ΦΩΤΙΣΜΟΣ

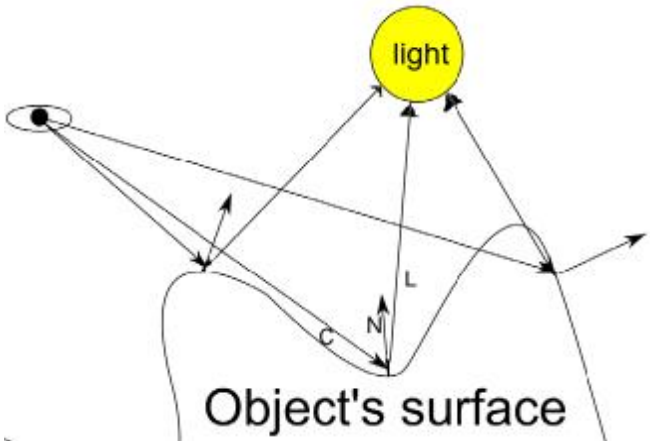
Ο φωτισμός είναι ένα αρκετά εκτεταμένο θέμα. Υπάρχουν τόσες πολλές τεχνικές, από επίπεδη σκίαση(flat shading) έως κανονική χαρτογράφηση(normal mapping), περνώντας από την απόφραξη του περιβάλλοντος(ambient occlusion) ή άλλες τεχνικές παγκόσμιου φωτισμού(global illumination). Θα μιλήσουμε σε αυτή την ενότητα σχετικά με τον τρόπο προετοιμασίας των vertex shader μας για να επιτύχουμε, στο επόμενο κεφάλαιο, πειστικές επιδράσεις φωτισμού χρησιμοποιώντας fragment shaders.

5.6.1 ΒΑΣΙΚΗ ΘΕΩΡΙΑ ΦΩΤΙΣΜΟΥ

Πρώτα απ 'όλα, πρέπει να καταστεί σαφές τι περιλαμβάνει το μοντέλο φωτισμού Phong, γιατί σε αυτό βασίζονται όλα τα παραδείγματα μας και τίποτα δεν είναι καλύτερο από ένα διάγραμμα γι' αυτό το σκοπό.

Το μοντέλο φωτισμού Phong είναι μια απλή εξίσωση. Διαχωρίζει το φως σε τρία συστατικά:

• Περιβάλλον (ambient): Παρέχει σταθερή συμβολή φωτισμού.
• Διάχυση (diffuse): Αυτό παρέχει συμβολή φωτισμού που εξαρτάται από τη θέση του φωτός και το σημείο της επιφάνειας που φωτίζεται (συνήθως αντιπροσωπεύεται από ένα κανονικό διάνυσμα).
•Κάτοπτρο (specular): Αυτό παρέχει συμβολή φωτισμού που εξαρτάται από τη θέση του φωτός, το σημείο της επιφάνειας (normal) και τη θέση της κάμερας / θεατή. Αυτό το στοιχείο φωτισμού είναι μια μικρή και φωτεινή κουκκίδα που εμφανίζεται σε πολύ ανακλαστικές επιφάνειες, όπως μέταλλο ή πλαστικό.



Στις σκηνές μας, θα έχουμε συνήθως ένα σημείο φωτός, μια κάμερα (το μάτι) και ένα αντικείμενο που θα ανάψει. Για να φωτίζουμε κάθε σημείο της επιφάνειας του αντικειμένου, πρέπει να γνωρίζουμε τουλάχιστον ένα σημαντικό πράγμα: το κανονικό διάνυσμα σε αυτό το ακριβές σημείο (ένα κανονικό διάνυσμα είναι ένα κανονικοποιημένο διάνυσμα που είναι κάθετο σε σημεία της επιφάνειας). Ας ονομάσουμε αυτό το διάνυσμα N. Χρειαζόμαστε επίσης το διάνυσμα που προέρχεται από το φως και φθάνει στο σημείο της επιφάνειας, καθώς και το διάνυσμα που προέρχεται από τη θέση της κάμερας (το μάτι στο διάγραμμα).

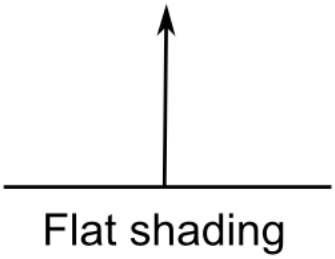
Σύμφωνα με αυτό το μοντέλο φωτισμού, οι περιοχές όπου τα κανονικά διανύσματα είναι παράλληλα με τα διανύσματα φωτός θα λάβουν μέγιστη επιρροή φωτισμού και οι περιοχές όπου η γωνία μεταξύ του κανονικού διανύσματος και του διανύσματος φωτός είναι μεγαλύτερη από 90 μοίρες δεν επηρεάζονται από το φως.

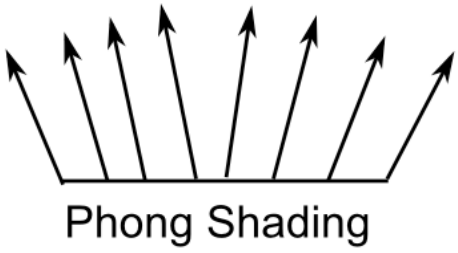
Η θέση της κάμερας επηρεάζει μόνο το κατοπτρικό στοιχείο του φωτός. Όταν ολοκληρώσουμε την εξίσωση φωτισμού, αυτό θα είναι σαφές.

Το κύριο μέλημά μας και η απόφαση που θα καθορίσει τον shader φωτισμού μας θα είναι ο τρόπος υπολογισμού του normal σε κάθε επιφάνεια.

Ανάλογα με τον τρόπο με τον οποίο υπολογίζουμε τους κανονικούς συντελεστές των τριγώνων, θα έχουμε διαφορετικές αποδόσεις:

• **Flat Shading**: Αυτό σημαίνει ότι έχουμε το ίδιο normal για ολόκληρη την επιφάνεια του τριγώνου και το ίδιο normal ισοδυναμεί με τον ίδιο φωτισμό σε ολόκληρη την επιφάνεια, έτσι ώστε το τρίγωνο να φωτίζεται ομοιόμορφα.
• **Σκίαση Gouraud**: Αυτό χρησιμοποιεί μόνο την συμβολή φωτισμού του υπολογισμού του normal κορυφής στον vertex shader και παρεμβάλλει αυτή τη συμβολή στον fragment shader.
• **Σκίαση Phong**: Χρησιμοποιούνται μόνο normal κορυφές, οι οποίες παρεμβάλλονται διαμέσου της επιφάνειας του τριγώνου στο fragment shader . Στη συνέχεια, ο φωτισμός υπολογίζεται στον fragment shader χρησιμοποιώντας παρεμβαλλόμενα normals.
• **Κανονική χαρτογράφηση(normal mapping)**: Τα normal προέρχονται από ένα χάρτη υφής. Έτσι, παίρνοντας ένα normal από μια υφή, έχουμε ένα πολύ ακριβές normal για κάθε fragment και όχι μόνο μια προσέγγιση με παρεμβολή.





5.6.2 ΚΩΔΙΚΑΣ ΠΑΡΑΔΕΙΓΜΑΤΟΣ ΦΩΤΙΣΜΟΥ

Τώρα για να ολοκληρώσουμε με τη θεωρία φωτισμού, ας εισαγάγουμε την εξίσωση φωτισμού. Στα γραφικά υπολογιστή, το φως συχνά χωρίζεται σε τρία (ή περισσότερα) ανεξάρτητα στοιχεία που συμβάλλουν στο τελικό αποτέλεσμα. Αυτά τα στοιχεία είναι, όπως προαναφέρθηκε, η συνεισφορά του περιβάλλοντος, η διάχυτη συμβολή και η κατοπτρική συμβολή.

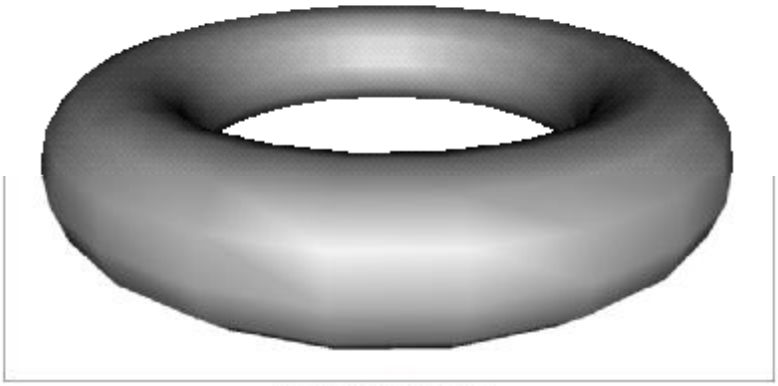
Και τα τρία αυτά στοιχεία υπολογίζονται και αθροίζονται μαζί για να επιτευχθεί η τελική συμβολή φωτισμού σε κάθε σημείο. Τώρα, ας γράψουμε ένα vertex shader για να παράγουμε ένα εφέ φωτισμού Gouraud:

```
   // Vertex shader
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   layout (location = 0) in vec4 Position;
   // Vertex normals buffer mapping
layout (location = 1) in vec3 VertexNormal;
   // Interpolator for our light
   smooth out vec3 lightContribution;
   /* Matrix to transform normals. This is the transpose of the
   inverse of the upper leftmost 3x3 of the modelview matrix */
   uniform mat3 NormalMatrix;
```

```
   uniform mat4 ModelView;
   uniform mat4 Projection;
   // We need to know where the light emitter is located
   uniform vec3 LightPosition;
   // The color of our light
   uniform vec3 LightColor;
   // Note: All vectors must always be normalized
   void main()
   {
   gl_Position = ProjectionModelView * Position;
   vec3 normal = normalize(NormalMatrix * VertexNormal);
   // Compute vertex position in camera coordinates
   vec4 worldVertexPos = ModelView * Position;
   // Compute the vector that comes from the light
   vec3 lightVec = normalize(LightPosition – worldVertexPos.xyz);
   /* Compute the vector that comes from the camera. Because we are
   working in camera coordinates, camera is at origin so the
   calculation would be:
   normalize(vec3(0,0,0) - worldVertexPos.xyz); */
   vec3 viewVec = normalize(-worldVertexPos.xyz);
   /* Calculate the specular contribution. Reflect is a built-in
   function that returns the vector that is the reflection of a
   vector on a surface represented by its normal */
   vec3 reflectVec = reflect(-lightVec, normal);
   float spec = max(dot(reflectVec, viewVec), 0.0);
   spec = pow(spec, 16.0);
   vec4 specContrib = lightColor * spec;
   /* We don't want any ambient contribution, but
   let's write it down for teaching purposes */
vec3 ambientContrib = vec3(0,0,0);
   // Calculate diffuse contribution
   vec3 diffContrib = lightColor * max(dot(lightVec, normal), 0);
   /* Final light contribution that will be interpolated in the
   fragment shader */
   lightContribution = ambientContrib + diffContrib + specContrib;
   }
   // fragment shader
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   // Interpolator for our light
   smooth int vec3 lightContribution;
   out vec4 frameBufferColor;
   void main()
   {
   frameBufferColor = vec4(lightContribution, 1.0);
}
```

Αυτός ο shader, που ονομάζεται επίσης φωτισμός ανά κορυφή, παράγει την ακόλουθη

απόδοση:



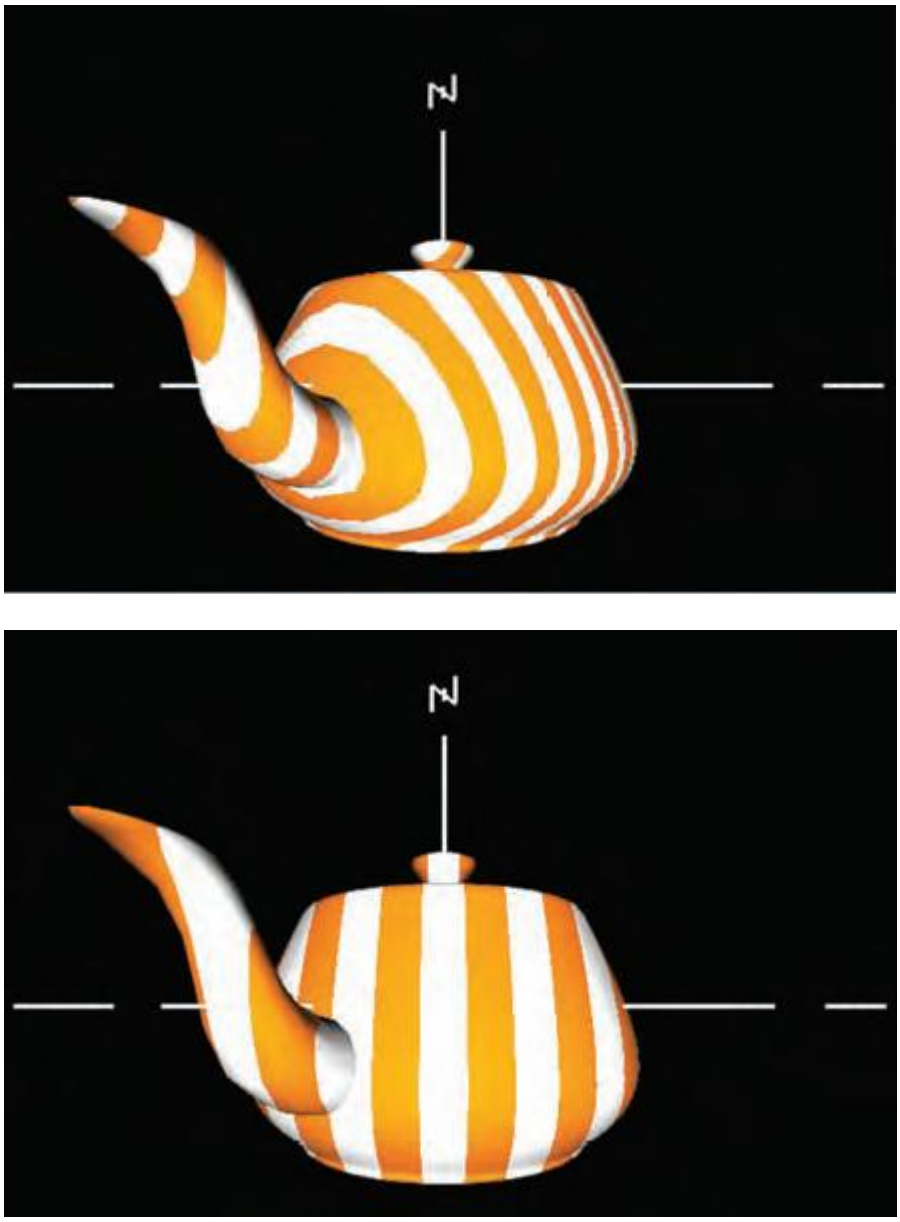
Gouraud lighting shader

Ας δούμε τώρα ακόμα ένα σύντομο παράδειγμα vertex shader προκειμένου να μπορέσουμε να κατανοήσουμε καλύτερα τον τρόπο αλλά και τα αποτελέσματα της χρήσης του.

5.7 ΠΑΡΑΔΕΙΓΜΑ ΧΡΗΣΗΣ VERTEX SHADER

(Mike Bailey)Αν αποφασίσουμε να χρησιμοποιήσουμε υπολογισμένα χρώματα ή υφές για την τελική εικόνα, με βάση τις συντεταγμένες κορυφών των γραφικών των αντικειμένων, είναι σημαντικό για τον vertex shader να ενεργοποιήσουμε αυτές τις συντεταγμένες ώστε να μεταφερθούν στον fragment shader, έτσι ώστε να μπορούν να χρησιμοποιηθούν εκεί . Υπάρχουν δύο είδη γεωμετρίας που μπορούμε να χρησιμοποιήσουμε για αυτό: γεωμετρία μοντέλου-χώρου(model-space geometry) ή γεωμετρία ματιού-χώρου(eye-space geometry). Χρησιμοποιούμε μια σύμβαση προθέματος για να τις δείξουμε. Το πρόθεμα MC αντιστοιχεί στη γεωμετρία του model-space, ενώ το πρόθεμα EC αντιστοιχεί στη γεωμετρία του eye-space. Αυτοί είναι, φυσικά, δύο από τους κύριους χώρους 3D που δουλεύουμε με τα γραφικά υπολογιστών. Μπορούμε να υπολογίσουμε αυτά τα κύρια είδη γεωμετρίας ως εξής.

• Για γεωμετρία model-space, απλά χρησιμοποιούμε τον χώρο στον οποίο ορίστηκε το μοντέλο μας: `vec3 MCposition = aVertex.xyz;`
• Για συντεταγμένες eye space, θέλουμε να εργαστούμε με τη γεωμετρία αφού έχουμε εφαρμόσει όλα τα μοντέλα. Αυτό είναι απλό με τη χρήση του πίνακα ModelView: `vec3 ECposition = (uModelViewMatrix * aVertex) .xyz;`



5.1 Η τσαγιέρα με συντεταγμένες μοντέλου που καθορίζουν τα χρώματα (πάνω) και με τις συντεταγμένες του ματιού που καθορίζουν τα χρώματα (κάτω).

Στο παραπάνω σχήμα , βλέπουμε πώς ένας shader μπορεί να χρησιμοποιήσει τις συντεταγμένες μοντέλου (πάνω) ή τις τιμές των συντεταγμένων του ματιού (κάτω) για να δημιουργήσει χρώματα. Ο fragment shader και για τις δύο εικόνες δημιουργεί ρίγες που είναι παράλληλες με το επίπεδο YZ, αλλά οι vertex shaders διαφέρουν στην αποστολή είτε συντεταγμένων μοντέλου είτε συντεταγμένων ματιού στον fragment shader που θα χρησιμοποιηθεί για τον προσδιορισμό των χρωμάτων. Η γεωμετρία και στις δύο περιπτώσεις έχει περιστραφεί για να δείξει ότι οι συντεταγμένες του μοντέλου παραμένουν με τη γεωμετρία του αντικειμένου, αλλά οι συντεταγμένες του ματιού παραμένουν σταθερές σε σχέση με τον χώρο προβολής. Δηλαδή, στην πάνω εικόνα, οι λωρίδες είναι παράλληλες προς το επίπεδο YZ των συντεταγμένων του μοντέλου και στην κάτω οι λωρίδες είναι παράλληλες με το επίπεδο YZ του περιστρεφόμενου (ματιού) συντεταγμένου χώρου.

Παρακάτω είναι ο shader κορυφής για το παραπάνω σχήμα, με ένα διακόπτη Boolean για να επιλέξουμε αν θέλουμε να στείλουμε τις συντεταγμένες του eye space ή του model space στον fragment shader. Ο υπολογισμός φωτισμού σε αυτόν τον shader είναι πολύ απλός, απλά με το να χειριστούμε την ένταση διάχυτου φωτός που θα αποτελούσε μέρος ενός μοντέλου πλήρους φωτισμού.

```
uniform bool uUseModelCoords;
out vec4 vColor;
out float vX, vY, vZ;
out float vLightIntensity;
void
main( )
{
vec3 TransNorm = normalize( uNormalMatrix * aNormal );
vec3 LightPos = vec3( 0., 0., 10. );
vec3 ECposition = ( uModelViewMatrix * aVertex ).xyz;
vLightIntensity = dot(normalize(LightPos - ECposition),
TransNorm);
vLightIntensity = abs( vLightIntensity );
vColor = aColor;
vec3 MCposition = aVertex.xyz;
if( uUseModelCoords )
{
vX = MCposition.x;
vY = MCposition.y;
vZ = MCposition.z;
}
else
{
vX = ECposition.x;
vY = ECposition.y;
vZ = ECposition.z;

}
gl_Position = uModelViewProjectionMatrix * aVertex;
}
```

Αυτός ο shader κορυφής δείχνει άλλες χρήσιμες τεχνικές. Λαμβάνει το χρώμα του αντικειμένου από τη μεταβλητή χαρακτηριστικού `aColor` και το μεταδίδει ως νέα μεταβλητή που θα χρησιμοποιηθεί από τον fragment shader, υπολογίζει την ένταση του φωτός χρησιμοποιώντας μια τυποποιημένη τεχνική διάχυτου φωτισμού και το περνά επίσης, ώστε ο φωτισμός να μπορεί να χρησιμοποιηθεί στον fragment shader. Ωστόσο, εάν θέλουμε να χρησιμοποιήσουμε το πλήρες μοντέλο φωτισμού ADS, πρέπει να λάβουμε υπόψη πολύ περισσότερα στοιχεία από την ένταση του φωτός.

6 FRAGMENT SHADERS

(Rodriguez)Οι fragment shaders είναι ίσως ο πιο σημαντικός τύπος από όλους τους shaders.
Η χρήση των fragment shaders είναι ο μόνος τρόπος να ζωγραφίσουμε κάτι στην οθόνη. Είναι υπεύθυνοι για την ενημέρωση του framebuffer με χρώματα και βάθος. Έτσι, εν συντομία, στην τελευταία φάση, όλα όσα βλέπουμε στην οθόνη ζωγραφίζονται από ένα fragment shader.

Σε αυτό το κεφάλαιο θα δούμε πώς εκτελούνται οι fragment shaders, τι μπορούν να κάνουν και τι δεν μπορούν. Επίσης, θα δούμε πώς αλληλεπιδρούν οι fragment shaders με άλλα στάδια του αγωγού.

6.1 ΜΟΝΤΕΛΟ ΕΚΤΕΛΕΣΗΣ

Όταν ένα στάδιο πρωτοκόλλου έχει τελειώσει, η επεξεργασία του στα στάδια πρωτοκόλλου κορυφής (vertex shaders , geometry shaders και clipping) ραστεροποιείται. Η εκτέλεση του fragment shader ξεκινά αυτή τη διαδικασία της ραστεροποίησης.

Εξετάζουμε ένα τρίγωνο. Για να ζωγραφίσουμε αυτό το τρίγωνο πάνω στην οθόνη θα πρέπει να το μετατρέψουμε από την φυσική διανυσματική μορφή του (συντεταγμένες κορυφών) σε διακριτά εικονοστοιχεία. Το σύστημα που εκτελεί τη συγκεκριμένη ενέργεια είναι το fragment shader module, το οποίο εκτελείται μία φορά ανά κάθε fragment.

Όσο περισσότερη από την περιοχή του framebuffer καλύπτει το πρωτόκολλο, τόσο περισσότερες φορές θα εκτελεστεί ο fragment shader. Η εκτέλεση των fragment shaders έχει άμεση επίδραση στο ποσοστό πληρώσεως της εφαρμογής μας (η ταχύτητα με την οποία μπορεί να παράγει fragment η GPU).

Ας δούμε δύο πολύ σημαντικούς περιορισμούς σχετικά με τους fragment shaders:

• Ένας fragment shader δεν μπορεί να διαβάσει το framebuffer. Είναι μόνο ικανός να γράψει σε αυτό. Εάν χρειάζεται να αναμίξουμε το framebuffer με το αποτέλεσμα των υπολογισμών του fragment shader μας, έχουμε μόνο δύο επιλογές και οι δύο συνεπάγονται τις τεχνικές πολλαπλου rendering:

°° Βασιζόμαστε στο μηχανισμό ανάμειξης για να αναμίξουμε το προηγούμενο περιεχόμενο του framebuffer με την τρέχουσα απόδοση(rendering).
°° Ρεντάρουμε πρώτα σε μια υφή και χρησιμοποιούμε αυτή την υφή ως είσοδο για το fragment shader.

• Ένας fragment shader εκτελείται για ένα συγκεκριμένο fragment που βρίσκεται σε μια συγκεκριμένη θέση του framebuffer (x, y). Με άλλα λόγια, δεν μπορούμε να χρησιμοποιήσουμε το shader για να τροποποιήσουμε άλλα μέρη του framebuffer.

6.2 ΤΕΡΜΑΤΙΖΟΝΤΑΣ ΕΝΑ FRAGMENT SHADER

Ο συνηθισμένος τρόπος τερματισμού της εκτέλεσης ενός fragment shader είναι όταν βρεθεί μια εντολή `return` ή όταν φτάσουμε στην αγκύλη κλεισίματος της λειτουργίας `main`, αλλά υπάρχει και ένας άλλος, ειδικά για ένα fragment shader. Εάν χρησιμοποιήσουμε τη λέξη-κλειδί `discard`, η εκτέλεση του fragment shader στο σημείο αυτό θα σταματήσει και ο framebuffer δεν θα ενημερωθεί καθόλου. Αυτό σημαίνει ότι δεν θα ενημερωθούν ούτε τα buffer χρώματος, βάθους και stencil ούτε οποιοσδήποτε άλλος buffer που αποτελεί μέρος του framebuffer. Άλλοι subbuffers, όπως για παράδειγμα το stencil buffer, θα παραμείνουν επίσης ανέγγιχτοι. Θα μπορούσαμε να απορρίψουμε τα fragments δημιουργώντας τρύπες στο πλέγμα μας.

6.3 ΕΙΣΟΔΟΙ ΚΑΙ ΕΞΟΔΟΙ

Όπως και οι υπόλοιποι shaders, οι fragment shaders μπορούν να λάβουν ομοιόμορφες μεταβλητές ως είσοδο. Λειτουργούν ακριβώς όπως για τους vertex shaders, οπότε δεν υπάρχουν πολλά να πούμε σε αυτό το σημείο εκτός αν μιλάμε για υφές. Οι υφές αντιπροσωπεύονται από έναν αδιαφανή μεταβλητό τύπο: έναν δειγματολήπτη(sampler) (ένας αδιαφανής τύπος είναι ένας ειδικός τύπος που μπορεί να χρησιμοποιηθεί μόνο με ενσωματωμένες λειτουργίες και δεν μπορεί να χρησιμοποιηθεί σε μαθηματικές πράξεις). Για να δημιουργήσουμε μια υφή, πρέπει να δηλώσουμε την ομοιόμορφη μεταβλητή της, χρησιμοποιώντας το σωστό τύπο sampler. Ανάλογα με την κατηγορία της υφής θα πρέπει να χρησιμοποιήσουμε τον ένα ή τον άλλο sampler.

Μια σύντομη λίστα είναι:

- `sampler1D` (για υφές 1D)
- `sampler2D` (για υφές 2D)
- `sampler3D` (για υφές 3D)
- `samplerCube` (για cubemaps)
- `sampler2DArray` (για συστοιχίες υφών 2D)
- `sampler2DShadow` (για χάρτες σκίασης 2D)

Οι samplers είναι ειδικοί μεταβλητοί τύποι που μπορούν να χρησιμοποιηθούν μόνο ως ενιαίοι(uniforms).Δεν μπορούμε να δηλώσουμε έναν sampler μέσα σε μια λειτουργία.

Στην εφαρμογή υποδοχής, όταν θέλουμε να φορτώσουμε ένα sampler, πρέπει απλά να ανεβάσουμε έναν ενιαίο τύπο `integer`, που αντιπροσωπεύει τη μονάδα υφής όπου δεσμεύεται η υφή προς το παρόν. Ένα κοινό λάθος είναι η χρήση του ονόματος υφής (ID) αντί της μονάδας υφής. Ένα μικρό δείγμα θα ήταν:

```
/* To textureNames is an array of strings and ids with
name of the texture inside the shader  */
for(int i = 0; i<numberOfTextures; ++i)
{
// Activate texture unit 'i'
glActiveTexture(GL_TEXTURE0 + i);
glBindTexture(GL_TEXTURE_2D, textureNames[i].id);
intloc          =          glGetUniformLocation(shaderID,
textureNames[i].name);
```

```
glUniform1i(loc, i);
}
```

Όπως αναφέρθηκε στο Κεφάλαιο Vertex Shaders , τα fragment shaders λαμβάνουν τα χαρακτηριστικά κορυφής (ή άλλες υπολογιζόμενες τιμές σε βάση ανά κορυφή) που παρεμβάλλονται με έναν από τους διαθέσιμους τρόπους (για εμάς, γραμμικά και προοπτικά διορθωμένα), χάρη στους διαχωριστές.

Για να διευκρινιστεί, ο όρος interpolator(παρεμβολέας) δεν είναι επίσημη λέξη στη GLSL, αλλά μια περιγραφή αυτού που κάνουν. Ανάμεσα στις διαφορετικές εκδόσεις OpenGL, είχαν πάρει διαφορετικά ονόματα. Στις πρώτες εκδόσεις GLSL και στην τρέχουσα γλώσσα σκίασης OpenGL / ES,το όνομά τους είναι ποικίλες μεταβλητές(varying variables). Επί του παρόντος, στην OpenGL, υπάρχουν απλά οι μεταβλητές `in`, αλλά επειδή το `in` δεν είναι ένα πολύ καλό όνομα και η `varying` είναι ξεπερασμένη, θα αναφερθούμε σε αυτά ως interpolators.

Ως αποτελέσματα(εξόδους), θα αναφέρουμε δύο. Κατά τη διάρκεια εκτέλεσης ενός fragment shader πρέπει να εξάγουμε το χρώμα του τρέχοντος fragment. Αυτό είναι υποχρεωτικό και το κύριο καθήκον ενός fragment shader, αλλά μπορούμε προαιρετικά να τροποποιήσουμε το βάθος του fragment.

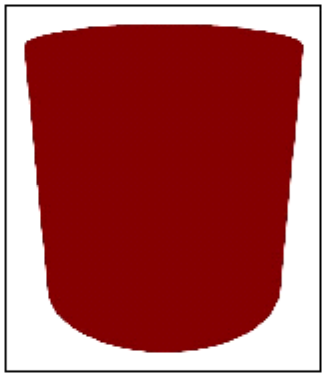
Για να τροποποιήσουμε το βάθος του fragment, δεν χρειάζεται να δηλώσουμε τίποτα, επειδή υπάρχει ήδη μια ειδική ενσωματωμένη μεταβλητή για το σκοπό αυτό: η `gl_FragDepth`. Γράφοντας εκείνη τη μεταβλητή (είναι απλώς μια μεταβλητή float) θα τροποποιήθεί το βάθος του fragment. Αυτό είναι πολύ βολικό για μερικά εφέ, όπως τα HUDs, 3D rendering γραμματοσειράς, κορώνες, φωτοβολίδες, κ.α., αλλά όχι για κανονικά renderings, οπότε σπάνια θα τα τροποποιήσουμε στα παραδείγματα.

6.4 ΣΥΜΠΑΓΕΣ ΠΛΕΓΜΑ ΧΡΩΜΑΤΟΣ (SOLID COLOR MESH)

Ο πρώτος είναι ένας πολύ απλός shader που γράφει το πλέγμα σε ένα συμπαγές χρώμα, που λαμβάνεται χρησιμοποιώντας μια ομοιόμορφη μεταβλητή:

```
   #version 430
   #debug(on)
   #optimize(off)
   uniform vec4 SolidColor;
   out vec4 FBColor;
   void main()
   {
   FBColor= SolidColor;
}
```

Χρησιμοποιώντας ένα μισό κόκκινο χρώμα που είναι αποθηκευμένο στην ομοιόμορφη μεταβλητή `SolidColor`, σε ένα κυλινδρικό πλέγμα, έχουμε την ακόλουθη απόδοση:



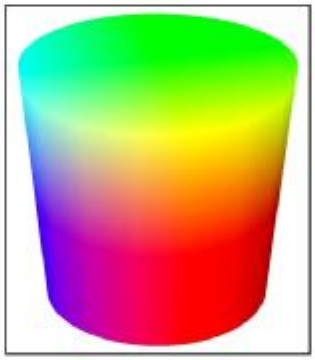
Χρησιμοποιώντας μόνο uniforms(ομοιόμορφα) ως εισροές δεν μπορούμε να κάνουμε πολλά περισσότερα από αυτό, γι 'αυτό θα ζωγραφίσουμε τον κύλινδρο χρησιμοποιώντας τις θέσεις των δικών του κορυφών.

6.5 ΠΑΡΕΜΒΑΛΛΟΜΕΝΟ ΧΡΩΜΑΤΙΣΜΕΝΟ ΠΛΕΓΜΑ (INTERPOLATED COLORED MESH)

Αρχικά, στον vertex shader, ορίζουμε ένα interpolator τύπου vec3 και τοποθετούμε το `xyz` του χαρακτηριστικού `position` εκεί. Στη συνέχεια, χρησιμοποιούμε τον ακόλουθο fragment shader:

```
   #version 430
   #pragma debug(on)
   #pragma optimize(off)
   uniform vec4 SolidColor;
   in vec3 InterpolatedPosition;
   out vec4 FBColor;
   void main()
   {
   FBColor = vec4(InterpolatedPosition, 1.0);
}
```

Ο παρακάτω είναι ο χρωματισμένος κύλινδρος ανά κορυφή ως αποτέλεσμα του rendering με αυτόν τον τελευταίο shader:



Τα χρώματα απεικονίζονται στην περιοχή 0 έως 1, όπου 0 είναι η ελάχιστη ένταση και 1 είναι η μέγιστη ένταση. Οι ενδιάμεσες τιμές αντιπροσωπεύουν τις ενδιάμεσες εντάσεις. Ένα πλήρες χρώμα αποτελείται από τέσσερα κανάλια: κόκκινο, πράσινο, μπλε και άλφα (διαφάνεια) στην αντίστοιχη σειρά. Για παράδειγμα, το (1, 0, 0, 1) αντιπροσωπεύει αδιαφανές καθαρό κόκκινο, το (1, 1, 0, 1) αδιαφανές καθαρό κίτρινο, και το (0,5, 0,5, 0,5, 0,5) ημι-διαφανές γκρι.

Ίσως εργαστήκαμε στο παρελθόν με χρώματα στην περιοχή 0 έως 255. Είναι το ίδιο πράγμα, και η μόνη διαφορά είναι ότι μέσα σε ένα shader, τα χρώματα κανονικοποιούνται αυτόματα, και το πιο σημαντικό, είναι συμπτυγμένα. Επομένως, εάν λόγω ενός υπολογισμού, ένα χρώμα γίνει -1,5 ή 2,9, συμπτύσσεται σε 0 ή 1.

Ας δούμε με βάση τα παραπάνω τι συνέβη με τον κύλινδρο μας. Επειδή οι συντεταγμένες των κορυφών βρίσκονται σε αυθαίρετο εύρος, ίσως μεγαλύτερο από 1 και με αρνητικές τιμές, συμπτύσσονται στο `[0,1]`, πράγμα που σημαίνει ότι κάθε τιμή κορυφής μεγαλύτερη από 1 θα είναι κορεσμένη όταν χρησιμοποιείται ως χρώμα και δεδομένου ότι ο κύλινδρος είναι κεντραρισμένος `(0,0,0)`, τα χρώματα είναι ακτινωτά. Έτσι, επειδή στο δείγμα μας το RGB είναι στην ίδια σειρά με το XYZ, το `R` αντιπροσωπεύει την κατεύθυνση x, το `G` την κατεύθυνση y και το `B` την κατεύθυνση z, και αυτό εξηγεί γιατί ο κύλινδρος μας παίρνει αυτά τα χρώματα.

6.6 ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ ΤΟΥΣ INTERPOLATORS ΓΙΑ ΝΑ ΥΠΟΛΟΓΙΣΟΥΜΕ ΤΙΣ ΣΥΝΤΕΤΑΓΜΕΝΕΣ ΥΦΗΣ

Χρησιμοποιώντας τον vertex shader του προηγούμενου κεφαλαίου που ενεργοποίησε έναν interpolator για συντεταγμένες υφής, πρόκειται να χαρτογραφήσουμε μια υφή πάνω στον κύλινδρο μας χρησιμοποιώντας τον ακόλουθο fragment shader:

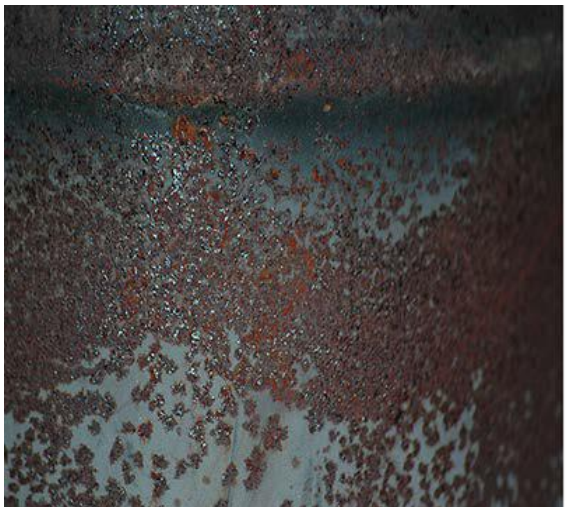
```
#version 430
#pragma debug(on)
#pragma optimize(off)
uniform sampler2D Image;
in vec2 TexCoordInterpolated;
```

```
   out vec4 FBColor;
   void main()
   {
   /* Next line means: "take the texel (texture pixel) of
   'Image' at 'TexCoordInterpolated' position"  */
   FBColor = texture(Image, TexCoordInterpolated);
}
```

Όπως βλέπουμε, έχουμε χρησιμοποιήσει μια ενσωματωμένη λειτουργία. Στην περίπτωση αυτή, `texture`. Υπάρχουν πολλές λειτουργίες αναζήτησης υφής, ανάλογα με τη χρήση που κάνουμε.

Επίσης, δηλώσαμε ένα αναγνωριστικό(identifier) για την υφή. Αυτή είναι μια μεταβλητή τύπου sampler και μπορεί να χρησιμοποιηθεί μόνο σε λειτουργίες αναζήτησης υφής.

Τα μοντέλα με υφή θα μοιάζουν με την ακόλουθη απόδοση:



6.7 ΦΩΤΙΣΜΟΣ PHONG

Συνεχίζουμε με το shader φωτισμού που παρουσιάζεται στο κεφάλαιο Vertex Shaders, αλλά αυτή τη φορά, θα περάσουμε όλους τους υπολογισμούς στο fragment shader.

Ο στόχος του επόμενου shader θα είναι να επιτευχθεί φωτισμός ανά εικονοστοιχείο, αλλά χρησιμοποιώντας μια μάσκα αποθηκευμένη στο άλφα κανάλι της υφής για να εξουδετερώσει την κατοπτρική συμβολή του φωτός σε αυτές τις κόκκινες σκουριασμένες περιοχές.

Επιπλέον, πρόκειται να χρησιμοποιήσουμε ένα πιο λεπτομερές 3D μοντέλο. Ο πιο συνηθισμένος σκοπός για να προσθέσουμε φωτισμό σε μια τρισδιάστατη σκηνή είναι να δώσουμε έμφαση στις λεπτομέρειες της επιφάνειας. Ένα αεροπλάνο ή ένας κύλινδρος δεν έχει λεπτομέρειες επιφάνειας επειδή είναι ομοιόμορφα, επομένως δεν θα βελτιωθούν οι λεπτομέρειες με τη χρήση φωτισμού εκεί.

Ας ξεκινήσουμε λοιπόν με ένα δείγμα σκίασης Phong:

```
   // Vertex shader for simple Phongs Lighting model
   #version 430
   #pragma optimize(off)
   #pragma debug(on)
   uniform mat4 Modelview;
   uniform mat4 Projection;
   /* Matrix to transform normals. This is the transpose of the
   inverse of the upper leftmost 3x3 of the modelview matrix */
   uniform mat3 NormalMatrix;
   layout (location = 0) in vec3 Position;
   layout (location = 1) in vec2 TextCoord;
   layout (location = 2) in vec3 Normal;
   smooth out vec2 TextCoordInterp;
   smooth out vec3 PositionInterp;
   /*This time we will interpolate the Normal, not the total computed
   lighting contribution */
   smooth out vec3 NormalInterp;
   void main()
   {
   PositionInterp = (Modelview * vec4(Position, 1.0)).xyz;
   TextCoordInterp = TextCoord;
   // Normals transform
   NormalInterp = normalize(NormalMatrix * Normal);
   gl_Position = Projection * Modelview * vec4(Position, 1.0);
}
```

Η μόνη διαφορά με τον shader που επεξηγήθηκε στο προηγούμενο κεφάλαιο είναι ότι, επιπλέον, παρεμβαίνουμε τη θέση των κορυφών. Αυτή η παρεμβαλλόμενη τιμή, στον fragment shader θα είναι η ισχύουσα θέση 3D στις συντεταγμένες της κάμερας (επειδή το μετασχηματίσαμε χρησιμοποιώντας τον πίνακα μοντέλου) του fragment κάθε τρίγωνου. Χρειαζόμαστε αυτή την τιμή επειδή πρόκειται να υπολογίσουμε τον φωτισμό σε αυτό ακριβώς το σημείο:

```
   // fragment shader for simple Phongs Lighting model
   #version 430
   #pragma optimize(off)
#pragma debug(on)
   uniform sampler2D Image;
   // Light position, in camera coordinates
   uniform vec3 LightPosition;
   // The color of our light
   uniform vec3 LightColor;
   smooth in vec2 TextCoordInterp;
   smooth in vec3 NormalInterp;
   smooth in vec3 PositionInterp;
```

```
   out vec4 FBColor;
   void main()
   {
   /* After interpolation, normals probably are denormalized,
   so we need renormalize them */
   vec3 normal = normalize(NormalInterp);
   /*Calculate the rest of parameters exactly like we did in the
   vertex shader version of this shader */
   vec3 lightVec = normalize(LightPosition - PositionInterp);
   vec3 viewVec = normalize(-PositionInterp);
   vec3 reflectVec = reflect(-lightVec, normal);
   float spec = max(dot(reflectVec, viewVec), 0.0);
   spec = pow(spec, 16.0);
   vec4 textureColor = texture(Image, TextCoordInterp);
   vec3 specContrib = LightColor * spec;
   // No ambient contribution this time
   vec3 ambientContrib = vec3(0.0, 0.0, 0.0);
   vec3 diffContrib = LightColor * max(dot(lightVec, normal), 0.0);
   /* Apply the mask to the specular contribution. The "1.0 –" is
   To invert the texture's alpha channel  */
   vec3  lightContribution  =  ambientContrib  +  diffContrib  +
   (specContrib * (1.0 – textureColor.a));
   FBColor = vec4(textureColor.rgb * lightContribution, 1.0);
}
```

Όπως βλέπουμε, ο shader είναι κατά 90 τοις εκατό ίσος με την έκδοση φωτός ανά κορυφή(per-vertex), αλλά η προκύπτουσα απόδοση διαφέρει αρκετά.

Θυμόμαστε ότι η κανονικοποίηση του κανονικού διανύσματος ξανά στο fragment shader, μετά την παρεμβολή, θα απορυθμιστεί. Επίσης, όλα τα άλλα διανύσματα πρέπει επίσης να κανονικοποιηθούν. Αυτό είναι πολύ σημαντικό, επειδή τα εσωτερικά γινόμενα που συμμετέχουν στους υπολογισμούς χρειάζονται τις κανονικοποιημένες τιμές για να λειτουργούν σωστά.

Θα παρουσιάσουμε μερικές παραλλαγές αυτού του shader για να δούμε τι συμβαίνει στο εσωτερικό όταν οι φωτισμοί, η υφή και οι μάσκες εφαρμόζονται προοδευτικά.

Πρώτον, ας δούμε το αποτέλεσμα όταν δεν εφαρμόζεται καθόλου φωτισμός και εφαρμόζεται μόνο υφή:

Δεν είναι πολύ λεπτομερές. Λοιπόν, ας δούμε τι συμβαίνει όταν αντικαθιστούμε την υφή από τους υπολογισμούς φωτισμού, χρησιμοποιώντας το ίδιο ακριβώς πλέγμα:

Αυτό έχει περισσότερο ενδιαφέρον. Παρατηρούμε τις κατοπτρικές αντανακλάσεις, τις πιο φωτεινές, και το ομαλό σκούρο χρώμα στις περιοχές όπου το φως μόλις φτάνει.

Για να επιτύχουμε αυτό το φαινόμενο φωτισμού, μεταβαίνουμε στο fragment shader και απλά αφαιρούμε την αναζήτηση υφής και αντικαθιστούμε τις δύο τελευταίες γραμμές με τα εξής:

```
   vec3 lightContribution = ambientContrib + diffContrib +
   specContrib;
FBColor = vec4(vec3(lightContribution), 1.0);
```

Τώρα, ας δούμε πώς η μάσκα που αποθηκεύεται στο κανάλι άλφα κρύβει την κατοπτρική συμβολή στην επόμενη απόδοση:

Αν θέλουμε αυτό το αποτέλεσμα, απλώς πολλαπλασιάζουμε την κατοπτρική συμβολή με ένα μείον το άλφα κανάλι της υφής (1.0 - textureColor.a).

Τώρα, ας συνδυάσουμε τον φωτισμό με τη μάσκα και την υφή, που είναι το ακριβές αποτέλεσμα του shader που εξηγήσαμε νωρίτερα:

Παρατηρούμε εδώ πώς οι κόκκινες σκονισμένες περιοχές σκιάζουν την κατοπτρική συμβολή, καθιστώντας αυτό το μοντέλο πιο ρεαλιστικό.

Σήμερα, το Phong είναι ένα βασικό μοντέλο φωτισμού, αλλά πριν από μερικά χρόνια ήταν ένα προηγμένο μοντέλο, επειδή οι κάρτες γραφικών είχαν αρκετή ισχύ επεξεργασίας μόνο γι 'αυτό. Τώρα, οι shaders μπορούν να είναι μεγαλύτεροι και πιο περίπλοκοι, και μάλιστα μπορούν να έχουν ένθετους βρόχους(nested loops) ή βρόχους με μεταβλητό αριθμό επαναλήψεων (προερχόμενοι από μια ομοιόμορφη μεταβλητή ή υφές)από ότι οι σταθερές. Μπορούν να έχουν υποδιαιρέσεις, λειτουργίες, δεκάδες προσπελάσεις υφής και πολλά άλλα πράγματα.

Σήμερα θα μπορούσε κανείς να προγραμματίσει έναν ανιχνευτή ακτίνων μέσα σε ένα shader ή ακόμα και σε άλλο Παγκόσμιο Μοντέλο Φωτισμού(Global Illumination Lighting Model) όπως είναι η Ambient Occlusion ή η Radiosity. Η ισχύς είναι εκεί για να χρησιμοποιήσουμε.

6.8 ΦΩΤΕΙΝΟΤΗΤΑ

(Mike Bailey)Ανεπίσημα, η φωτεινότητα μπορεί να θεωρηθεί ως η ποσότητα του μη-μαύρου σε ένα χρώμα. Για το χρώμα RGB, το "λιγότερο" μαύρο σημαίνει ότι τα στοιχεία χρώματος είναι πλησιέστερα στο 1.0 και το "περισσότερο" μαύρο σημαίνει ότι τα στοιχεία είναι πλησιέστερα στο 0.0. Για να χειριστούμε τη φωτεινότητα μιας εικόνας, χρησιμοποιούμε μια μαύρη εικόνα με χρώμα

```
target = vec3(0.0, 0.0, 0.0)
```

ως βάση. Οι τιμές του *uT* κάτω του 1.0 θα σκουρύνουν κάθε συστατικό του χρώματος, ενώ οι τιμές μεγαλύτερες του 1.0 θα φωτίζουν κάθε συστατικό του χρώματος μέχρι το σημείο όπου το χρώμα είναι σταθερό. Αυτό μπορεί, φυσικά, να "ξεπλύνει" τα χρώματα εάν τα χρώματα είναι ήδη φωτεινά ή αν χρησιμοποιούμε πολύ μεγάλο μέγεθος *uT*. Αυτό φαίνεται στην εικόνα 6.1.

Το δείγμα κώδικα για έναν πολύ απλό fragment shader που ρυθμίζει τη φωτεινότητα φαίνεται παρακάτω. Στην πραγματικότητα, η φωτεινότητα της εικόνας γίνεται με την αφαίρεση του μαύρου από αυτήν.

```
uniform sampler2D uImageUnit;
uniform float uT;
in vec2 vST;
out vec4 fFragColor;
void main( )
{
vec3 irgb = texture( uImageUnit, vST ).rgb;
vec3 black = vec3( 0., 0., 0. );
fFragColor = vec4( mix( black, irgb, uT ), 1. );
}
```

Εικόνα 6.1 Φωτισμός σε φωτογραφία από προϊστορικό γαλλικό τάφο με uT = 0.0 (αριστερή), 1.0 (μεσαία) και 2.0 (δεξιά).

6.9 ΑΝΤΙΘΕΣΗ (CONTRAST)

Η αντίθεση σε μια εικόνα περιγράφει πόσα χρώματα ξεχωρίζουν από το γκρι. Για να

χειριστούμε την αντίθεση σε μια εικόνα, χρησιμοποιούμε ως βασική εικόνα μια σταθερή εικόνα γκρι 50%, η οποία υπολογίζεται εύκολα

```
target = vec3(0.5,0.5,0.5);
```

Οι τιμές παραμέτρων του *T* που είναι μικρότερες του 1 θα μετακινήσουν κάθε στοιχείο χρώματος προς το 0.5, μειώνοντας την αντίθεση στην εικόνα, ενώ οι τιμές μεγαλύτερες από 1 θα μετακινήσουν κάθε στοιχείο χρώματος μακριά από το 0.5, αυξάνοντας την αντίθεση, όπως φαίνεται στην εικόνα 6.2.

Το δείγμα κώδικα για ένα πολύ απλό fragment shader που ρυθμίζει τη φωτεινότητα ή την αντίθεση φαίνεται παρακάτω. Στην πραγματικότητα, η φωτεινότητα της εικόνας γίνεται με αφαίρεση του μαύρου από αυτήν και η αντίθεση αυξάνεται αφαιρώντας το 50% του γκρι από αυτήν.

```
#define BRIGHTNESS

#undef CONTRAST
uniform sampler2D uImageUnit;
uniform float uT;
in vec2 vST;
our vec4 fFragColor;
void main( )
{
vec3 irgb = texture( uImageUnit, vST ).rgb;

#ifdef BRIGHTNESS
vec3 target = vec3( 0., 0., 0. );
#else
vec3 target = vec3( 0.5,0.5,0.5 );
#endif
fFragColor = vec4( mix( target, irgb, uT ), 1. );
}
```

Εικόνα 6.2 Διαχείρηση της αντίθεσης σε μια φωτογραφία μιας κατεστραμμένης γαλλικής μονής με T = 0.0 (αριστερή), 1.0 (μεσαία) και 2.5 (δεξιά).

7 GEOMETRY SHADERS

(Rodriguez)Σε αυτό το κεφάλαιο, θα γνωρίσουμε έναν νέο τύπο shader ο οποίος θα χρησιμεύσει στη βελτίωση των σκηνών μας, δίνοντάς τους περισσότερη ευελιξία ενώ μας επιτρέπει να αλληλεπιδράσουμε πιο βαθιά με τον αγωγό, αλλά χωρίς να χρειαστεί να αυξήσουμε τον πλευρικό κώδικα της CPU ή το εύρος ζώνης μεταξύ της CPU και της GPU.

Θα μάθουμε πώς λειτουργούν αυτοί οι νέοι shaders ενώ θα αναλύσουμε μερικά παραδείγματα τεχνικών όπου οι geometry shaders μπορούν να μας δώσουν μεγάλη ευελιξία. Θα καλύψουμε τα ακόλουθα θέματα:

• Πώς να αναπαράγουμε τη γεωμετρία εν πτήσει, για παράδειγμα, για να δημιουργήσουμε δίδυμα μοντέλα

• Πώς να φτιάξουμε ένα σύστημα σωματιδίων ή πλήθους με πολύπλοκα σωματίδια διαφόρων σχημάτων που διαχωρίζονται από άδεια σημεία

7.1 ΟΙ GEOMETRY SHADERS ΣΕ ΣΧΕΣΗ ΜΕ ΤΟΥΣ VERTEX SHADERS

Αν και οι geometry shaders μετασχηματίζουν τις κορυφές με πολύ παρόμοιο τρόπο σε σχέση με τους vertex shaders, πρέπει να απαριθμήσουμε κάποιες σημαντικές διαφορές μεταξύ τους:

• Οι vertex shaders εκτελούνται μία φορά από μια εισερχόμενη κορυφή. Οι geometry shaders εκτελούνται μία φορά (από προεπιλογή) από ένα εισερχόμενο πρωτόκολλο.

• Οι vertex shaders δεν μπορούν να έχουν πρόσβαση σε πληροφορίες σχετικά με γειτονικές κορυφές. Ένας geometry shader έχει όλες τις πληροφορίες για ένα δεδομένο πρωτόκολλο και άλλα γειτονικά.

• Οι vertex shaders δεν παράγουν νέες κορυφές. Οι geometry shaders δημιουργούν νέα πρωτόκολλα (στην πραγματικότητα, αυτός είναι ο κύριος σκοπός).

Επιπλέον, υπάρχουν και άλλα στοιχεία σχετικά με τους geometry shaders που πρέπει να είναι γνωστά:

• Το στάδιο geometry shader συμβαίνει μετά το στάδιο συναρμολόγησης πρωτοκόλλου.

• Ένας geometry shader λαμβάνει συγκεντρωμένα πρωτόκολλα. Αυτό σημαίνει ότι δεν έχει σημασία αν στέλνουμε τρίγωνα, ταινίες τριγώνων(strips) ή βεντάλιες τριγώνων(fans). Ένας geometry shader θα λαμβάνει πάντα ένα τρίγωνο. Το ίδιο συμβαίνει και με τις γραμμές, τις λωρίδες γραμμής(strips) ή τους βρόχους γραμμών(loops). Τα πρωτόκολλα που θα φτάσουν στον geometry shader θα είναι απλές γραμμές.

• Το εισερχόμενο πρωτόκολλο θα απορριφθεί μετά την εκτέλεση του geometry shader.

• Ο τύπος του πρωτοκόλλου εισόδου δεν είναι υποχρεωτικό να είναι η έξοδος. Αυτό σημαίνει ότι, για παράδειγμα, μπορούμε να χρησιμοποιήσουμε σημεία ως εισόδους και ταινίες τριγώνου(strips) ως εξόδους.

Οι geometry shaders είναι διαθέσιμοι ως επέκταση της OpenGL από σχεδόν οποιαδήποτε έκδοση της OpenGL, αλλά από την έκδοση 4.0, ένας geometry shader μπορεί να χρησιμοποιηθεί περισσότερο από μία φορά ανά πρωτόκολλο. Μπορούμε να χρησιμοποιήσουμε μια ενσωματωμένη μεταβλητή `(gl_InvocationID)` για να ξεχωρίζουμε μεταξύ κάθε εκτέλεσης στο ίδιο πρωτόκολλο κατά τη διάρκεια της ίδιας σχεδιασμένης κλήσης.

Υπάρχουν επίσης πολλές πρόσθετες έξοδοι για έναν geometry shader και αντί για τα πρωτόκολλα (viewport ID ,layer ID), αλλά αυτά προορίζονται για προχωρημένους χρήστες.

7.2 ΕΙΣΟΔΟΙ ΚΑΙ ΕΞΟΔΟΙ

Εκτός από τις συνηθισμένες εισόδους και εξόδους (ιδιότητες κορυφών και uniforms), υπάρχουν νέα στοιχεία που πρέπει να ληφθούν υπόψη.

Τα πρωτόκολλα, το κύριο στοιχείο εισόδου και εξόδου μας, είναι πολύπλοκα και γι 'αυτό πρέπει να είμαστε προσεκτικοί και να καθορίσουμε πολύ καλά τι κάνουμε κάθε φορά. Ο πρωταρχικός τύπος που καθορίζεται στον geometry shader πρέπει να ταιριάζει ακριβώς με το πρωτόκολλο που σχεδιάζουμε (για παράδειγμα, χρησιμοποιώντας κλήση GL όπως `glDrawArrays`). Αυτό σημαίνει ότι δεν μπορούμε να γράψουμε με ευκολία έναν γενικού σκοπού geometry shader για όλα. Επίσης, πρέπει να καθορίσουμε τον πρωταρχικό τύπο και τον αριθμό των κορυφών που θα εκπέμπονται (δημιουργούνται) στατικά στον shader.

Για αυτούς τους σκοπούς, έχουμε τη λέξη-κλειδί `layout`. Με αυτή, θα μπορούσαμε να καθορίσουμε μια διάταξη(layout) εισόδου και εξόδου. Για παράδειγμα, θα μπορούσαμε να κάνουμε κάτι παρόμοιο με τα ακόλουθα:

```
layout(triangles) in;
layout(triangle_strip, max_vertices = 6) out;
```

Οι επιλογές για μια διάταξη εισόδου είναι οι εξής:

- `points`
- `lines` (αυτό συμπεριλαμβάνει πρωτόκολλα που σχεδιάζονται ως `GL_LINES`, `GL_LINE_STRIP`, και `GL_LINE_LOOP`)
- `lines_adjacency`[γειτνίαση γραμμών](`GL_LINES_ADJACENCY` `GL_LINE_STRIP_ADJACENCY`)
- `triangles` (συμπεριλαμβανομένων των `GL_TRIANGLES`, και `GL_TRIANGLE_STRIP`, και `GL_TRIANGLE_FAN`)
- `triangle_adjacency`[γειτνίαση τριγώνων] (συμπεριλαμβανομένων των `GL_TRIANGLES_ADJACENCY` και `GL_TRIANGLE_STRIP_ADJACENCY`)

Οι επιλογές για ένα επίπεδο εξόδου είναι οι εξής:

- `points`
- `line_strip`

- `triangle_strip`

Λαμβάνουμε υπόψη ότι στην περίπτωση απλών τριγώνων, οι μεμονωμένες γραμμές δεν μπορούν να χρησιμοποιηθούν ως έξοδος. Αν θέλουμε να εξάγουμε περισσότερες από μία γραμμές ή περισσότερα από ένα τρίγωνα, πρέπει και στις δύο περιπτώσεις να έχουν τη μορφή λωρίδων(strips). Επίσης, οι τύποι εισόδου και εξόδου δεν σχετίζονται, όπως θα δούμε στα παραδείγματα αυτού του κεφαλαίου.

Επίσης, πρέπει να ορίσουμε σταθερά τον αριθμό των κορυφών που θα εκπέμπονται (`max_vertices`).

7.3 ΜΠΛΟΚ ΔΙΑΣΥΝΔΕΣΗΣ (INTERFACE BLOCKS)

Με την ενσωμάτωση ενός νέου τύπου shaders, η επικοινωνία μεταξύ όλων γίνεται λίγο περίπλοκη. Τώρα δεν είναι αρκετοί οι βασικοί παρεμβολείς(interpolators) αν και η βασική ιδέα είναι ακόμα η ίδια. Ένας νέος μηχανισμός που ονομάζεται μπλοκ διασύνδεσης χειρίζεται αυτή την επικοινωνία με πιο αποτελεσματικό τρόπο.

Θα πρέπει να σκεφτούμε ένα μπλοκ διασύνδεσης ως δομές (αν και δεν είναι πραγματικά) που χρησιμεύουν για τη διασύνδεση στα στάδια των shaders. Οι ορισμοί του μπλοκ διασύνδεσης πρέπει να ταιριάζουν στους shaders όπου δηλώνονται, αν όχι, θα υπάρξει σφάλμα σύνδεσης.

Σε αυτό το σημείο, έχουμε δύο μεταξύ τους διαστήματα: μεταξύ των vertex και geometry shaders και μεταξύ των geometry και των fragment shaders. Για την πρώτη περίπτωση, πρέπει να ορίσουμε ένα μπλοκ διασύνδεσης εξόδου στον vertex shader και το ίδιο μπλοκ διασύνδεσης στον geometry shader αλλά ως μπλοκ εισόδου. Ας δώσουμε ένα παράδειγμα:

```
// vertex shader output interface block
out MyPerVertexVariables
{
smooth vec2 TextCoords;
smooth vec3 Normal;
} perVertex;
// geometry shader input interface block
in MyPerVertexVariables
{
smooth vec2 TextCoords;
smooth vec3 Normal;
} perVertex[]; // here the instance name is an array, to hold all perVertex of a primitive
```

Στην περίπτωση της επικοινωνίας μεταξύ geometry και fragment shaders, είναι λίγο πολύ το ίδιο:

```
// Geometry shader output interface block
out MyInterpolators
{
smooth vec2 TextCoords;
```

```
   smooth vec3 Normal;
   }interpolators;
   // fragment shader input interface block
   in MyInterpolators
   {
   smooth vec2 TextCoords;
   smooth vec3 Normal;
}interpolators;
```

Όπως μπορούμε να δούμε, αυτό είναι λίγο πολύ το ίδιο με τη χρήση κανονικών μεταβλητών εισόδου / εξόδου, αλλά υπάρχουν μερικοί λογικοί περιορισμοί (ο shader δεν μπορεί να ορίσει ένα μπλοκ διασύνδεσης εξόδου, ένας vertex shader δεν μπορεί να ορίσει ένα μπλοκ διασύνδεσης εισόδου).

Το κλειδί που επιτρέπει στη GLSL να εκτελέσει τη σύνδεση μεταξύ δύο μπλοκ διασύνδεσης είναι ότι το όνομα struct (όχι η μεταβλητή!) πρέπει να είναι το ίδιο. Στο παράδειγμά μας, οι `MyInterpolators`, `MyPerVertexVariables` και τα πεδία που ορίζονται στο εσωτερικό πρέπει να ταιριάζουν ακριβώς στα δύο στάδια του shader.

Το μεταβλητό όνομα του μπλοκ διασύνδεσης δεν επηρεάζει τη σύνδεση. Μπορεί να είναι διαφορετικό και στις δύο ομάδες.

Επίσης, η GLSL μας παρέχει ένα ενσωματωμένο μπλοκ διασύνδεσης που ονομάζεται `gl_in`. Προορίζεται για τη διατήρηση των θέσεων κορυφών ενός πρωτοκόλλου, και έχει οριστεί με αυτόν τον τρόπο:

```
   in gl_PerVertex
   {
   vec4 gl_Position;
   float gl_PointSize;
   float gl_ClipDistance[];
} gl_in[];
```

Οι μεταβλητές `gl_PointSize` και `gl_ClipDistance` δεν έχουν μεγάλο ενδιαφέρον για εμάς αυτήν τη στιγμή. Θα επικεντρωθούμε στο `gl_Position`.

Για να δώσουμε ένα παράδειγμα, θεωρούμε ότι οι τύποι εισόδου μας είναι τρίγωνα, μετά ότι το `gl_in` θα έχει μέγεθος 3 (ένα στοιχείο ανά κορυφή) και κάθε `gl_in []`. `Gl_Position` θα είναι η θέση αυτής της κορυφής αφού μετασχηματιστεί από τον vertex shader και συναρμολογηθεί κατά τη διάρκεια του αρχικού σταδίου συναρμολόγησης.

Μια τελευταία σημείωση σχετικά με τα μπλοκ διασύνδεσης είναι ότι μπορεί να είναι ανώνυμα. Δεν είμαστε υποχρεωμένοι να τους δώσουμε ένα (μεταβλητό) όνομα. Αν επιλέξουμε αυτόν τον τρόπο, η πρόσβαση στα μέλη είναι μόνο το όνομα μέλους. Σε μια κορυφή, οι shaders `gl_Position` δηλώνονται σιωπηρά ως:

```
   out gl_PerVertex
   {
   vec4 gl_Position;
   float gl_PointSize;
```

```
   float gl_ClipDistance[];
};
```

Καθώς το προηγούμενο μπλοκ διασύνδεσης δεν έχει όνομα εμφάνισης, μπορούμε να έχουμε άμεση πρόσβαση στα μέλη απευθείας, όπως κάναμε μέχρι τώρα. Μπορούμε επίσης να το κάνουμε αυτό με τα δικά μας μπλοκ διασύνδεσης, αλλά για λόγους σαφήνειας του κώδικα, είναι προτιμότερο να τα χρησιμοποιήσουμε με την ονοματολογία της συνήθους δομής (όνομα του μέλους δομής dot)

7.4 ΠΑΡΑΔΕΙΓΜΑ - ΠΕΡΑΣΜΑ ΑΠΟ ΤΟ SHADER

Ας γράψουμε τον πρώτο geometry shader μας, ένα πέρασμα που θα παράγει ως έξοδο το ίδιο πρωτόκολλο που φτάνει στο geometry shader ως είσοδος:

```
   #version 430
#pragma optimize(off)
   #pragma debug(on)
   // Establish our primitive's input and output types, as well as
   the number of vertices that we will produce.
   layout(triangles) in;
   layout(triangle_strip, max_vertices = 3) out;
   void main()
   {
   // GLSL provides us with the .length() method for arrays. It tells
   us the number of elements in the vector.
   for(int i = 0; i < gl_in.length(); ++i)
   {
   // gl_in is the standard input interface block
   gl_Position = gl_in[i].gl_Position;
   // The next function, will emit the computed vertex to the
   pipeline
   EmitVertex();
   }
   // The next function will close the primitive
   EndPrimitive();
}
```

Το πέρασμα από τον shader δεν έχει καμία επίδραση, με την έννοια ότι η απόδοση δεν επηρεάζεται καθόλου. Δεν είναι πολύ χρήσιμο σε ένα πραγματικό περιβάλλον, αλλά είναι πολύ βολικό για να δείξει πώς παράγεται ένα νέο πρωτόκολλο.

Το να δημιουργηθεί ένα νέο πρωτόκολλο είναι τόσο απλό όσο το να γραφτούν όλες οι μεταβλητές μιας κορυφής (στην περίπτωση αυτή, μόνο `gl_Position`) και στη συνέχεια να κληθεί η λειτουργία `EmitVertex ()`. Αυτή η λειτουργία θα δημιουργήσει αποτελεσματικά μια νέα κορυφή με τα δεδομένα που έχουν συμπληρωθεί στις μεταβλητές `out`. Όταν τελειώσουμε με όλες τις κορυφές, μια απλή κλήση στη λειτουργία `EndPrimitive ()` θα κλείσει το πρωτόκολλο και θα είναι έτοιμο για περαιτέρω στάδια σωλήνωσης.

7.5 ΠΑΡΑΔΕΙΓΜΑ - ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ ΙΔΙΟΤΗΤΕΣ ΣΤΑ ΜΠΛΟΚ ΔΙΑΣΥΝΔΕΣΗΣ

Για το επόμενο παράδειγμα, θα χρειαστεί να χρησιμοποιήσουμε περισσότερα χαρακτηριστικά κορυφής από ότι τη θέση, οπότε θα πρέπει να δημιουργήσουμε ένα μπλοκ διασύνδεσης για να περάσουμε όλα αυτά στο geometry shader.

Ας το δούμε στο παρακάτω παράδειγμα:

```
   // Vertex shader
   #version 430
   #pragma optimize(off)
   #pragma debug(on)
uniform mat4 Modelview;
   uniform mat4 Projection;
   layout (location = 0) in vec3 Position;
   layout (location = 1) in vec2 TextCoord;
   layout (location = 2) in vec3 Normal;
   // Now, we will define in this block the interpolators
   out VertexData
   {
   smooth vec2 TextCoord;
   smooth vec3 Normal;
   } vertexOut;
   void main()
   {
   gl_Position = Projection * Modelview * vec4(Position, 1.0);
   vertexOut.TextCoord = TextCoord;
   vertexOut.Normal = Normal;
}
```

Τώρα, στο geometry shader, θα πάρουμε αυτές τις νέες τιμές χρησιμοποιώντας το μπλοκ σε στυλ συστοιχίας:

```
   #version 430
   #pragma optimize(off)
   #pragma debug(on)
   layout(triangles) in;
   layout(triangle_strip, max_vertices = 3) out;
   // The important point here is “VertexData”. It must match with
   the vertex shader version.
   // input interface block, from the vertex shader
   in VertexData
   {
   smooth vec2 TextCoord;
   smooth vec3 Normal;
   } vertexIn[]; // The name of the variable doesn't have to match
```

```
   with the vertex shader's version
   // Output interface block, going to the fragment shader
   out Interpolators
   {
   smooth vec2 TextCoord;
   smooth vec3 Normal;
   }interpolators;
   void main()
   {
   for(int i = 0; i < gl_in.length(); ++i)
{
   gl_Position = gl_in[i].gl_Position;
   interpolators.TextCoord = vertexIn[i].TextCoord;
   interpolators.Normal = vertexIn[i].Normal;
   EmitVertex();
   }
   EndPrimitive();
}
```

Θυμόμαστε πως όλες οι μεταβλητές εξόδου και τα μπλοκ διασύνδεσης που ορίζονται σε ένα geometry shader είναι ανά κορυφή.

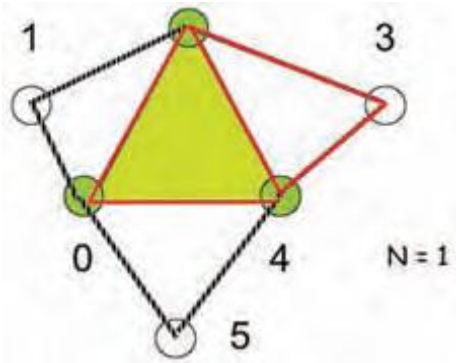
Στο fragment shader, παίρνουμε τις παρεμβαλλόμενες τιμές όπως πάντα:

```
   #version 430
   #pragma optimize(off)
   #pragma debug(on)
   uniform sampler2D Image;
   // From geometry shader or from vertex shader
   in Interpolators
   {
   smooth vec2 TextCoord;
   smooth vec3 Normal;
   }interpData;
   out vec4 FBColor;
   void main()
   {
   vec4 textureColor = texture2D(Image, interpData.TextCoord);
   // do something with interpData.normal;
   FBColor = vec4(textureColor.rgb, 1.0);
}
```

Τώρα έχουμε όλους τους βασικούς μηχανισμούς στα χέρια μας, ότι θα χρειαστεί για να γράψουμε ένα πλήρες χαρακτηριστικών geometry shader, οπότε ας πάμε με ένα shader που θα κάνει αυτό το είδος shader ενδιαφέρον.

7.6 ΦΙΓΟΥΡΕΣ 3D ΑΝΤΙΚΕΙΜΕΝΩΝ

(Mike Bailey)Ένας έξυπνος τρόπος για να ανιχνεύσουμε μια άκρη σε μια 3D φιγούρα είναι ότι μια άκρη φιγούρας μοιράζεται από γειτονικά τρίγωνα, με ένα που βλέπει προς το μάτι και το άλλο στραμμένο μακριά από το μάτι. Μπορούμε να προσδιορίσουμε τον τρόπο με τον οποίο κάθε τρίγωνο στρέφεται υπολογίζοντας το εσωτερικό γινόμενο της κανονικής επιφάνειας του τριγώνου και του διανύσματος προς το μάτι και να ελέγξουμε εάν το πρόσημό του είναι θετικό ή αρνητικό. Στο τρίγωνο με γειτνίαση που παρουσιάζεται στην εικόνα 7.1, αυτή η δοκιμή εφαρμόζεται στο κεντρικό τρίγωνο και σε κάθε ένα από τα τρίγωνα που γειτονεύουν με αυτό. Ένα τέτοιο ζεύγος τριγώνων επισημαίνεται στο σχήμα.



Εικόνα 7.1. Η δομή των τριγώνων που δίνει ένα τμήμα γραμμής της φιγούρας.

Εικόνα 7.2. Τρεις όψεις του λαγού, με ελάχιστο φωτισμό από κάτω, που δείχνει τις άκρες της φιγούρας.

Οι τύποι γεωμετρίας εισόδου και εξόδου για αυτόν τον geometry shader είναι τρίγωνα με γειτνίαση και line strips, αντίστοιχα. Η εντολή ***glman*** ObjAdj είναι παρόμοια με την εντολή Obj, αλλά αναλύει το αρχικό αρχείο Obj για να καθορίσει τις πληροφορίες γειτνίασης, έτσι ώστε τα πρωτόκολλα τριγώνου-με-γειτνίαση να είναι διαθέσιμα στον geometry shader.

Εδώ παραλείπονται τα αρχεία vertex και fragment shader. Ο vertex shader εκτελεί μόνο τον μετασχηματισμό ModelViewProjection και ο fragment shader ρυθμίζει μόνο το χρώμα pixel. Αυτά αποτελούν ρουτίνα.

Ο geometry shader λειτουργεί λαμβάνοντας ένα τρίγωνο με γειτνίαση και υπολογίζοντας την κανονική πρόσοψη σε κάθε ένα από τα τέσσερα τρίγωνα, εξασφαλίζοντας ότι κάθε normal δείχνει σωστά σύμφωνα με τις τυπικές συμβάσεις τριγώνου. Ο vertex shader έχει ήδη τοποθετήσει τις κορυφές στον τρισδιάστατο χώρο των ματιών, οπότε τα normals(μέσοι όροι) μπορούν να συγκριθούν με την απλή

σύγκριση των z στοιχείων τους. Εάν υπάρχει διαφορά σημείων μεταξύ του z στοιχείου του normal του κεντρικού τριγώνου και του z στοιχείου του normal ενός παρακείμενου τριγώνου, τότε η κοινή τους άκρη σχεδιάζεται εκπέμποντας δύο κορυφές και τερματίζοντας το πρωτόκολλο. Παρατηρούμε ότι κάθε άκρη του μεσαίου τριγώνου ελέγχεται επειδή, κατ 'αρχήν, η φιγούρα θα μπορούσε να περιλαμβάνει οποιοδήποτε από αυτά. Το αποτέλεσμα αυτού του shader φαίνεται στην εικόνα 7.2.

Geometry Shader silh.geom

```
#version 330
#extension GL_EXT_geometry_shader4: enable

layout( triangles_adjacency ) in;
layout( line_strip, max_vertices=32 ) out;
void main( )
{
vec3 V0 = gl_PositionIn[0].xyz;
vec3 V1 = gl_PositionIn[1].xyz;
vec3 V2 = gl_PositionIn[2].xyz;
vec3 V3 = gl_PositionIn[3].xyz;
vec3 V4 = gl_PositionIn[4].xyz;
vec3 V5 = gl_PositionIn[5].xyz;
vec3 N042 = cross( V4-V0, V2-V0 );
vec3 N021 = cross( V2-V0, V1-V0 );
vec3 N243 = cross( V4-V2, V3-V2 );
vec3 N405 = cross( V0-V4, V5-V4 );
// rashly assume all 4 normals are really meant to be
// within 90 degrees of each other:
if( dot( N042, N021 ) < 0. )
N021 = -N021;
if( dot( N042, N243 ) < 0. )
N243 = -N243;
if( dot( N042, N405 ) < 0. )
N405 = -N405;
// look for a silhouette edge between triangles 042 and
// 021:
if( N042.z * N021.z < 0. )
{
gl_Position = uProjectionMatrix* vec4( V0, 1. );
EmitVertex( );
gl_Position = uProjectionMatrix* vec4( V2, 1. );
EmitVertex( );
EndPrimitive( );
}
// look for a silhouette edge between triangles 042 and
// 243:
if( N042.z * N243.z < 0. )
{
gl_Position= uProjectionMatrix* vec4( V2, 1. );
EmitVertex( );
```

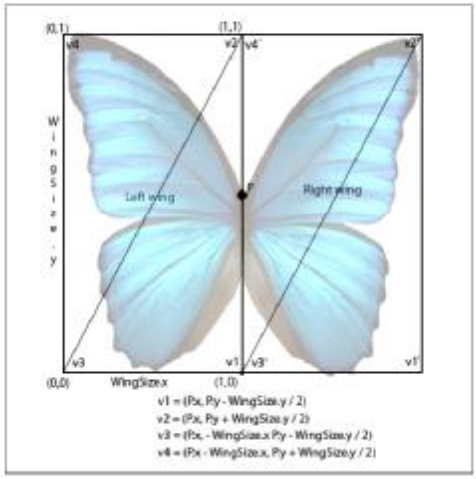
```
gl_Position= uProjectionMatrix* vec4( V4, 1. );
EmitVertex( );
EndPrimitive( );
}
// look for a silhouette edge between triangles 042 and
// 405:
if( N042.z * N405.z < 0. )
{
gl_Position= uProjectionMatrix* vec4( V4, 1. );
EmitVertex( );
gl_Position= uProjectionMatrix* vec4( V0, 1. );
EmitVertex( );
EndPrimitive( );
  }
}
```

7.7 ΕΝΑ ΠΛΗΘΟΣ ΠΕΤΑΛΟΥΔΩΝ

(Rodriguez)Σε αυτό το παράδειγμα, θα δούμε πώς να κάνουμε σύνθετες δομές, δημιουργώντας όλες τις πληροφορίες εν πτήσει.

Στόχος μας είναι να αποδώσουμε ένα πλήθος πεταλούδων που χρησιμοποιούν τη CPU όσο το δυνατόν λιγότερο με την πραγματοποίηση υπολογισμών (μετασχηματισμοί, άνοιγμα σημαίας, κλίση πεταλούδας / γήπεδο κλπ.). Θα ξεκινήσουμε από τυχαία κατανομή σημείων στο χώρο. Δεν θα χρησιμοποιηθούν άλλα χαρακτηριστικά κορυφής εκτός από τη θέση των σημείων και από αυτά θα δημιουργήσουμε για κάθε σημείο δύο ορθογώνια με υφή (τριγωνικές λωρίδες/strips), ένα για κάθε φτερό πεταλούδας. Θα αλλάξουμε επίσης τις διαδρομές των ανοιγμάτων των φτερών και των πεταλούδων που χρησιμοποιούν ξεχωριστές τιμές οι οποίες βασίζονται στην ενσωματωμένη μεταβλητή `gl_PrimitiveID`.

Για να συνοψίσουμε τον τρόπο με τον οποίο θα κατασκευάσουμε αυτές τις ταινίες τριγώνου από ένα μόνο σημείο, μια εικόνα αξίζει χίλιες λέξεις:



Με το P, το αρχικό μέγεθος σημείων, υπολογίζουμε τις γωνίες και των δύο ταινιών(strips) τριγώνου προσθέτοντας ή αφαιρώντας τις διαστάσεις των πτερυγίων που περνάμε ως ομοιόμορφη μεταβλητή.

Τώρα, ας ξεκινήσουμε με τον κώδικα:

1. Κατ 'αρχάς, ας γράψουμε τον vertex shader μας, ο οποίος, παρεμπιπτόντως, είναι ο πιο απλός vertex shader που θα μπορούσαμε να γράψουμε ποτέ:

```
#version 430
#pragma optimize(off)
#pragma debug(on)
layout (location = 0) in vec3 Position;
void main()
{
/ * This time we won't transform the position here,
but in the Geometry Shader  */
gl_Position = vec4(Position, 1.0);
}
```

2. Τώρα, για να βάλουμε όλα τα απλά πράγματα μαζί, ας δούμε το fragment shader, ο οποίος δεν εμπλέκεται πολύ επίσης:

```
#version 430
#pragma optimize(off)
#pragma debug(on)
uniform sampler2D Image;
// Interface block to pass the texture coordinates to the
fragment shader
```

```
in Interpolators
{
smooth vec2 TextCoord;
}interpData;
out vec4 FBColor;
void main()
{
vec4 textureColor = texture2D(Image, interpData.TextCoord);
/ * To render only the parts where the texture is not
transparent, we discard those fragments who are
transparent in some degree  * /
if(textureColor.a < 0.4)
{
discard;
}
FBColor = textureColor;
}
```

3. Τώρα, ας πάμε με τον πυρήνα του προγράμματος μας, τον geometry shader:

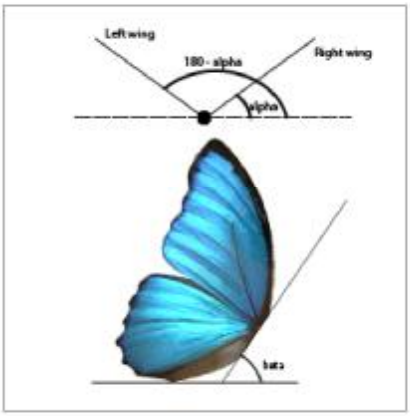
```
#version 430
#pragma optimize(off)
#pragma debug(on)
layout(points) in; // we feed the GS with points
/  *  The  output  will  be  two  separated  triangle  strips,  4
vertices each one. We will rotate each triangle strip using the
shared edge as pivot  * /
layout(triangle_strip, max_vertices = 8) out;
uniform vec2 WingSize;
uniform int NumberOfPrimitives; // total number of points
uniform mat4 Modelview;
uniform mat4 Projection;
out Interpolators
{
smooth vec2 TextCoord;
}interpData;
void main()
{
/ * Alpha angle is the wings aperture angle. This will give us
a rotation in the range [5, 85] degrees  * /
// gl_PrimitiveIDIn is the index of the current primitive that
is being processed .
float   alpha   =   radians(5.0f   +   (gl_PrimitiveIDIn   /
(NumberOfPrimitives - 1.0f)) * 80.0);
// Beta angle is the orientation of the butterfly
float   beta   =   radians(-90.0f   +   (gl_PrimitiveIDIn   /
(NumberOfPrimitives - 1.0f)) * 90.0f);
```

```
      // Matrix to translate the wing to the origin
      mat4 T = mat4(vec4(1,0,0,0),
      vec4(0,1,0,0),
      vec4(0,0,1,0),
      vec4(-gl_in[0].gl_Position.xyz,1));
      // Matrix to translate the wing back to its original position
      mat4 Ti = mat4(vec4(1,0,0,0),
      vec4(0,1,0,0),
      vec4(0,0,1,0),
      vec4(gl_in[0].gl_Position.xyz,1));
      / * Matrix to rotate the whole butterfly to change its flying
      direction. This is a Z axis rotation matrix.  * /
      mat4 Rz = mat4(vec4(cos(beta), sin(beta), 0, 0),
      vec4(-sin(beta), cos(beta), 0, 0),
vec4(0, 0, 1, 0),

      vec4(0, 0, 0, 1));
      // Left wing creation.
      // Matrix to rotate the left wing and give an appearance of
      moving wings. This is a Y axis rotation matrix
      mat4 Ry = mat4(vec4(cos(alpha), 0, -sin(alpha), 0),
      vec4(0, 1, 0, 0),
      vec4(sin(alpha), 0, cos(alpha), 0),
      vec4(0, 0, 0, 1));
      // Final transform matrix
mat4 M = Projection * Modelview * Rz * Ti * Ry * T;
```

4. Μέχρι τώρα, ετοιμάσαμε μόνο τους πίνακες για να πραγματοποιήσουμε τις περιστροφές με τον τρόπο που δείχνει το επόμενο διάγραμμα.

5. Το διάγραμμα τοποθέτησης πεταλούδας και διάτρησης φτερών που παίρνουμε είναι ως εξής:



6. Τώρα, ας υπολογίσουμε τις κορυφές. Η διαδικασία είναι πολύ απλή. Απλώς υπολογίζουμε τις θέσεις και τα άλλα χαρακτηριστικά κορυφής και βγάζουμε αυτήν την κορυφή. Όταν τελειώσουμε με όλες τις κορυφές του πρωτοκόλλου, το ολοκληρώνουμε
EndPrimitive ():

```
// Using the original point's position and the wing's size, we
calculate each new vertex. Then we transform it with the accumulated
matrix (M)
      gl_Position  =  M  *  vec4(gl_in[0].gl_Position.x,  gl_in[0].
      gl_Position.y - WingSize.y / 2.0f, gl_in[0].gl_Position.zw);
      // Create a proper texture coordinate for this vertex
      interpData.TextCoord = vec2(1.0f, 0.0f);
      EmitVertex();
      // 2nd vertex
      gl_Position  =  M  *  vec4(gl_in[0].gl_Position.x,  gl_in[0].
      gl_Position.y + WingSize.y / 2.0f, gl_in[0].gl_Position.zw);
      interpData.TextCoord = vec2(1.0f, 1.0f);
      EmitVertex();
      // 3rd vertex
      gl_Position = M * vec4(gl_in[0].gl_Position.x - WingSize.x,
      gl_in[0].gl_Position.y  -  WingSize.y  /  2.0f,  gl_in[0].
      gl_Position.zw);
      interpData.TextCoord = vec2(0.0f, 0.0f);
      EmitVertex();
      //4rd vertex
      gl_Position = M * vec4(gl_in[0].gl_Position.x - WingSize.x,
      gl_in[0].gl_Position.y  +  WingSize.y  /  2.0f,  gl_in[0].
      gl_Position.zw);
      interpData.TextCoord = vec2(0.0f, 1.0f);
      EmitVertex();
```

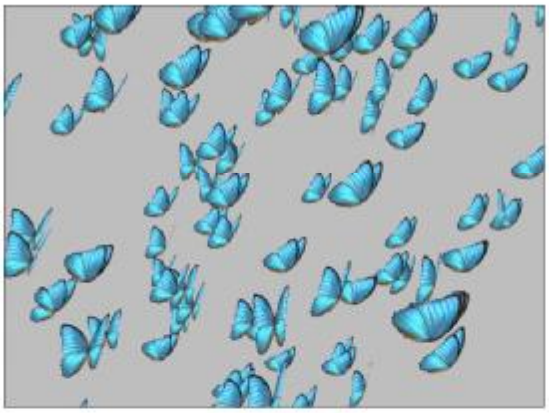
```
EndPrimitive();
```

7. Εδώ τελειώνουμε με το αριστερό φτερό. Τώρα, επειδή το δεξί φτερό έχει μια αντίθετη περιστροφή, πρέπει να υπολογίσουμε ένα νέο πίνακα περιστροφής και ένα νέο συγκεντρωμένο πίνακα μετασχηματισμού:

```
      // The aperture angle for the right wing is 180 - alpha
      alpha = 3.141592f - alpha;
      Ry = mat4(vec4(cos(alpha), 0, -sin(alpha), 0),
      vec4(0, 1, 0, 0),
      vec4(sin(alpha), 0, cos(alpha), 0),
      vec4(0, 0, 0, 1));
      M = Projection * Modelview * Rz * Ti * Ry * T;
      gl_Position = M * vec4(gl_in[0].gl_Position.x + WingSize.x,
      gl_in[0].gl_Position.y - WingSize.y / 2.0f, gl_in[0].
      gl_Position.zw);
      interpData.TextCoord = vec2(0.0f, 0.0f);
      EmitVertex();
      gl_Position = M * vec4(gl_in[0].gl_Position.x + WingSize.x,
      gl_in[0].gl_Position.y + WingSize.y / 2.0f, gl_in[0].
      gl_Position.zw);
      interpData.TextCoord = vec2(0.0f, 1.0f);
EmitVertex();
      gl_Position = M * vec4(gl_in[0].gl_Position.x, gl_in[0].
      gl_Position.y - WingSize.y / 2.0f, gl_in[0].gl_Position.zw);
      interpData.TextCoord = vec2(1.0f, 0.0f);
      EmitVertex();
      gl_Position = M * vec4(gl_in[0].gl_Position.x, gl_in[0].
      gl_Position.y + WingSize.y / 2.0f, gl_in[0].gl_Position.zw);
      interpData.TextCoord = vec2(1.0f, 1.0f);
      EmitVertex();
      EndPrimitive();
}
```

Αυτός ήταν ένας μακρύς shader, αλλά αν το διαβάσουμε, τα πιο περίπλοκα πράγματα είναι οι μετασχηματισμοί. Το υπόλοιπο, δημιουργώντας και συμπληρώνοντας τα χαρακτηριστικά των κορυφών, είναι αρκετά απλό.

Αυτό είναι το τελικό αποτέλεσμα αυτού του shader:



8 COMPUTE SHADERS

Έως τώρα, ακολουθήσαμε τους κανόνες σωλήνωσης απόδοσης(rendering pipeline). Τα δεδομένα στέλνονταν πάντα με τη μορφή buffer κορυφών ή υφής και μετασχηματίζονταν σύμφωνα με πολύ καλά καθορισμένα βήματα. Τώρα θα μάθουμε για ένα νέο μηχανισμό για τη χρήση της GPU με πιο ευέλικτο τρόπο με όλα τα πλεονεκτήματα της GLSL. Αυτό σημαίνει ότι, σε αντίθεση με τις άλλες λύσεις της GPU, θα έχουμε όλες τις διαθέσιμες μορφές OpenGL υφής (όχι ένα περιορισμένο υποσύνολο όπως στο CUDA ή OpenCL), τύπους διανυσμάτων και πινάκων και ενσωματωμένες λειτουργίες.

Ο compute shader είναι ένα πολύ εκτεταμένο θέμα, με πολλές λεπτομέρειες που δεν μπορούν να καλυφθούν σε ένα μόνο κεφάλαιο οποιουδήποτε βιβλίου. Για το λόγο αυτό, σε αυτό το κεφάλαιο, θα καλύψουμε μόνο τα βασικά, μια σύντομη εισαγωγή στη γλώσσας των υπολογιστών των shaders και τις δυνατότητες που θα μπορούσαν να μας προσφέρουν.

Σε αυτό το κεφάλαιο, θα μάθουμε πώς οι γενικές διαδικασίες της GPU είναι διατεταγμένες στον επεξεργαστή και πώς αυτές εκτελούνται. Δεν θα επικεντρωθούμε μόνο στην απόδοση(rendering) και τις τροποποιήσεις εικόνας, αλλά και στην παραγωγή αριθμητικών αποτελεσμάτων.

8.1 ΜΟΝΤΕΛΟ ΕΚΤΕΛΕΣΗΣ

Στην περίπτωση διαφορετικών τύπων shader, η σωλήνωση απόδοσης αποφάσισε πώς και πότε έπρεπε να εφαρμοστούν. Από τη στιγμή που η σωλήνωση απόδοσης σταμάτησε να λειτουργεί, έπρεπε να κανονίσουμε τη δουλειά της - συγχρονισμό, διαχείριση νήματος(thread) και ούτω καθεξής.

Θεωρούμε εκ των προτέρων την εκτέλεση ενός μόνο geometry shader. Η ορολογία των geometry shader ορίζει αυτό ως στοιχείο εργασίας. Αυτά τα στοιχεία εργασίας δεν μπορούν να χρησιμοποιηθούν ανεξάρτητα. Χωρίζονται σε ομάδες. Μια ομάδα εργασίας περικλείει έναν αυθαίρετο αριθμό στοιχείων εργασίας, ο οποίος καθορίζεται από τον προγραμματιστή αλλά περιορίζεται από τις δυνατότητες υλικού. Οι ομάδες εργασίας τρέχουν όλα τα αντικείμενα τους με παράλληλο τρόπο - περισσότερο ή λιγότερο συγχρονισμένα. Επίσης, τα στοιχεία μέσα σε μια ομάδα μπορούν να μοιράζονται μηχανισμούς μνήμης και συγχρονισμού. Όλες οι ομάδες εργασίας πρέπει να έχουν τον ίδιο αριθμό αντικειμένων εργασίας και κάθε ομάδα εργασίας εκτελείται ανεξάρτητα από τις υπόλοιπες, απομονωμένη και σε απροσδιόριστη σειρά.

Επιπλέον, οι ομάδες εργασίας ομαδοποιούνται ανάλογα με τις διαστάσεις. Μπορούμε να επιλέξουμε να τις οργανώσουμε σε μία, δύο ή τρεις διαστάσεις.

Η ρύθμιση των στοιχείων εργασίας και των ομάδων εργασίας χωρίζεται μεταξύ της εφαρμογής κεντρικού υπολογιστή και του πηγαίου κώδικα των shaders. Στην εφαρμογή κεντρικού υπολογιστή, κατά τη διάρκεια της κλήσης GL που καλεί τον compute shader, θα καθορίσουμε πόσες ομάδες εργασίας πρόκειται να χρησιμοποιήσουμε και στον πηγαίο

κώδικα του shader θα καθορίσουμε πόσες θέσεις εργασίας θα συνθέσουν μια ομάδα εργασίας.

Η διάσταση των ομάδων εργασίας έχει σκοπό να μας βοηθήσει να αναδείξουμε το πρόβλημά μας. Θα σκεφτόσαστε ότι όσο περισσότερες ομάδες εργασίας και στοιχεία εργασίας, τόσο το καλύτερο, διότι έτσι χρησιμοποιούμε περισσότερα τμήματα της GPU και αυτό θα αποδόσει με μεγαλύτερη επίδοση. Αυτό είναι σωστό, αλλά η απόδοση δεν είναι όλα. Αν και ο ίδιος αριθμός αντικειμένων εργασίας θα εκτελεστεί με τη διαμόρφωση δέκα ομάδων εργασίας με δέκα στοιχεία εργασίας (10 x 10 = 100) και όχι με τη διαμόρφωση πέντε ομάδων εργασίας στη διάσταση X και δύο ομάδων εργασίας στην διάσταση Y, με δέκα στοιχεία εργασίας ανά ομάδα (5 x 2 x 10 = 100), ο δείκτης του αντικειμένου εργασίας που θα εκτελεστεί θα μπορούσε να μας βοηθήσει πολύ, ανάλογα με τον τρόπο δομής του προβλήματος. Είναι σαν να επεξεργαζόμαστε μια ολόκληρη εικόνα που έχει πρόσβαση στα εικονοστοιχεία με έναν μονοδιάστατο τρόπο αντί της χρήσης των x και y για πρόσβαση στα εικονοστοιχεία.

Από το shader μπορούμε να έχουμε πρόσβαση σε πολύ χρήσιμες πληροφορίες: την ομάδα εργασίας όπου βρίσκεται το τρέχον αντικείμενο εργασίας, το αναγνωριστικό(ID) του τρέχοντος αντικειμένου εργασίας μέσα σε μια ομάδα εργασίας, το παγκόσμιο αναγνωριστικό του στοιχείου εργασίας (αναγνωριστικό του στοιχείου εργασίας πολλαπλασιασμένο με το μέγεθος της ομάδας εργασίας), και λίγα περισσότερα. Αυτά τα αναγνωριστικά(IDs) θα μας βοηθήσουν να αναδείξουμε συστοιχίες δεικτών, εικόνες (ή ακόμα και συστοιχίες εικόνων!), και buffers μιας, δύο και τριών διαστάσεων. Οι ενσωματωμένες μεταβλητές που είναι διαθέσιμες στη γλώσσα για να μας βοηθήσουν να διαχειριστούμε και να κανονίσουμε αυτά τα αναγνωριστικά είναι οι εξής:

```
   // Total number of work groups in each dimension
   in uvec3 gl_NumWorkGroups;
   // Number of work items inside a work group
   const uvec3 gl_WorkGroupSize;
   // Index of the work group that is currently in execution
   in uvec3 gl_WorkGroupID;
   // Index of the work item (inside a work group) that is currently
   in execution
   in uvec3 gl_LocalInvocationID;
   / * Global index of the current work item  (gl_WorkGroupID *
   gl_WorkGroupSize + gl_LocalInvocationID;) */
   in uvec3 gl_GlobalInvocationID;
   /*
  This is a one dimensional representation of   gl_LocalInvocationID
(gl_ LocalInvocationID.z *  gl_WorkGroupSize.x * gl_WorkGroupSize.y
   + gl_LocalInvocationID.y * gl_WorkGroupSize.x +
   gl_LocalInvocationID.x;)
   */
   uint gl_LocalInvocationIndex;
```

Αν όμως αντί να διασχίσουμε ένα μονοδιάστατο πίνακα έχουμε να διασχίσουμε μια εικόνα, μπορούμε να κανονίσουμε τα αντικείμενα εργασίας και τις ομάδες εργασίας να τα χρησιμοποιήσουν για να δείξουν τις δύο διαστάσεις της εικόνας και έτσι να ρυθμίσουν ή να

πάρουν ένα εικονοστοιχείο της εικόνας. Για παράδειγμα, εάν έχουμε μια εικόνα με μέγεθος 512 x 512, μπορούμε να πούμε ότι πρόκειται να χρησιμοποιήσουμε ομάδες εργασίας 32 αντικειμένων στο X και 32 αντικείμενα στο Y και να χρησιμοποιήσουμε 16 ομάδες εργασίας στο X και άλλες 16 σε Y (32 χ 16 = 512).

Ο αριθμός των διαθέσιμων αντικειμένων εργασίας και των ομάδων εργασίας περιορίζεται από το υλικό. Μπορούμε να ρωτήσουμε την OpenGL στην εφαρμογή υποδοχής για αυτά τα όρια.

8.2 ΑΠΟΔΟΣΗ ΠΑΡΑΔΕΙΓΜΑΤΟΣ ΥΦΗΣ

Το πρώτο παράδειγμα θα πάρει μια υφή και θα γράψει σε αυτό μερικά χρώματα. Για να επεξηγήσουμε τη χρήση ομάδων εργασίας και αντικειμένων εργασίας, θα ζωγραφίζουμε κάθε εικονοστοιχείο της εικόνας με το ευρετήριο της ομάδας εργασίας που επεξεργάζεται αυτό το εικονοστοιχείο. Καθώς θα χρησιμοποιήσουμε 16 x 16 ομάδες εργασίας, η εικόνα θα γεμίσει με 16 x 16 στερεά μπλοκ χρωμάτων.

Η κλήση shader γίνεται με αυτή τη γραμμή: `glDispatchCompute (16, 16, 1) ;`. Αυτό σημαίνει ότι θα εκτελέσουμε τον shader με 16 ομάδες εργασίας στις διαστάσεις X και Y και 1 στο Z (1 είναι η ελάχιστη αποδεκτή τιμή για μια διάσταση).

```
   #version 430
   // Configure how many work items compose the work groups
   // Because we are executing the shader with 16 x 16 work groups,
   we can handle images is 512 x 512 (16*32)
   layout(local_size_x = 32,local_size_y = 32, local_size_z = 1) in;
   // Declare the uniform variable that holds our image
   uniform layout(rgba8) writeonly image2D Image;
   void main()
   {
   ivec2 position = ivec2(gl_GlobalInvocationID.xy);
   vec4 color=vec4(gl_WorkGroupID / vec3(gl_NumWorkGroups), 1.0);
   imageStore(Image, position, color);
}
```

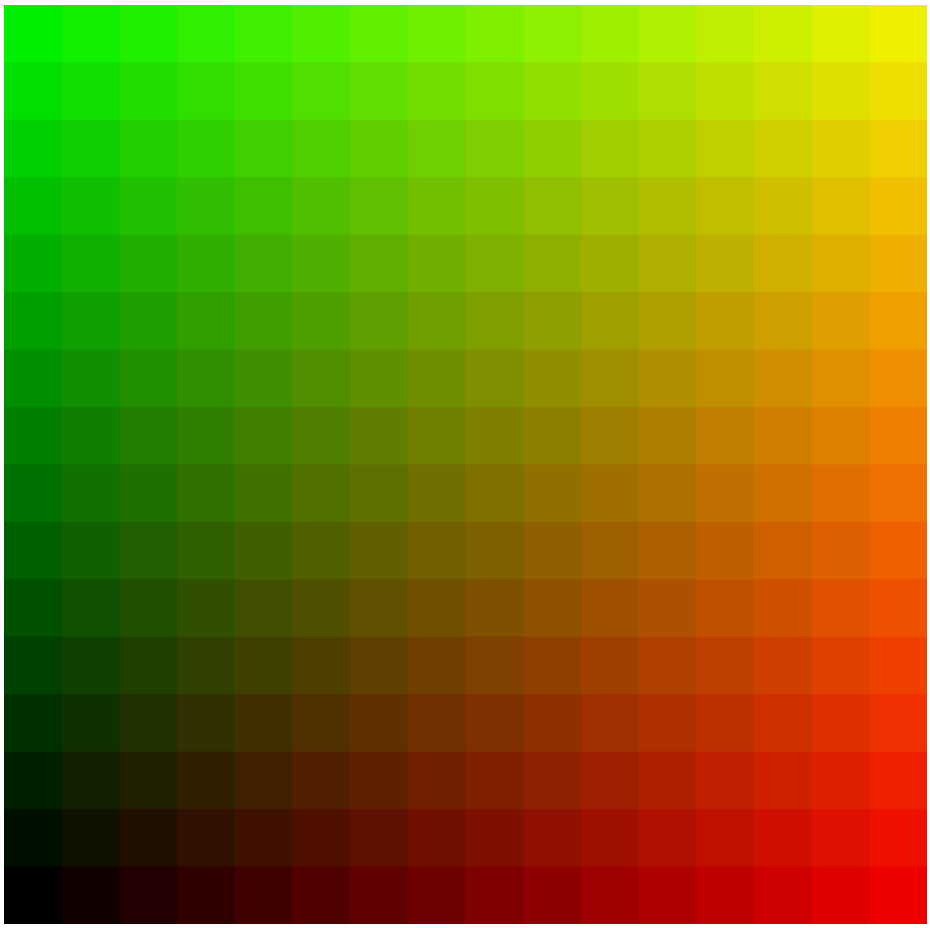
Υπάρχουν μερικά πράγματα που πρέπει να παρατηρήσουμε στις προηγούμενες γραμμές.

Πρώτον, μπορούμε να δούμε ότι ρυθμίζουμε το αντικείμενο εργασίας. Επίσης, μπορούμε να δούμε νέες λέξεις-κλειδιά στη δήλωση ενιαίου δειγματολήπτη(sampler).

Το πιο σημαντικό είναι ότι χρησιμοποιούμε το παγκόσμιο αναγνωριστικό ID (αν απαριθμούμε γραμμικά όλα τα στοιχεία εργασίας, το παγκόσμιο αναγνωριστικό ID θα είναι αυτό του δείκτη) του τρέχοντος αντικειμένου εργασίας. Σε αυτήν την περίπτωση, οι τιμές αυτές κυμαίνονται από 0 έως 511 για κάθε διάσταση (16 ομάδες εργασίας x 32 εργασίες = 512 εκτελέσεις shader), αλλά το πιο σημαντικό είναι στην επόμενη γραμμή: για να έχουμε πρόσβαση στην εικόνα, χρησιμοποιούμε μια νέα λειτουργία πρόσβασης υφής (`imageStore`) που παίρνει ολοκληρωμένες τιμές, κυμαίνεται από 0 έως μέγεθος υφής 1. Αυτό είναι πολύ

διαφορετικό από ό, τι στους κανονικούς shaders, όπου οι συντεταγμένες υφής ήταν μεταβλητές κυμαινόμενου σημείου και οι τιμές κανονικοποιήθηκαν. Αυτό γίνεται με αυτό τον τρόπο σε compute shaders εν μέρει για να αποφευχθεί οποιοδήποτε είδος φιλτραρίσματος, στρογγυλοποιήσεις, ή ανακρίβειες τη στιγμή της γραφής σε μια υφή. Πρέπει να καθορίσουμε το ακριβές texel στο οποίο θα έχουμε πρόσβαση.

Όπως ήταν αναμενόμενο, αυτός ο shader παρήγαγε ένα 16χ16 σχήμα μπλοκ, όπως αυτό:



Σχετικά με το τελευταίο παράδειγμα: τους compute shaders, όταν ασχολούμαστε με εικόνες, πρέπει να τους χειριζόμαστε με διαφορετικό τρόπο από ό, τι κάνουμε με τους vertex, fragment ή geometry shaders.

Ο τρόπος πρόσβασης και η μορφή της εικόνας πρέπει να ρυθμιστούν σύμφωνα με τον ακόλουθο πίνακα:

Float image formats (Φορτωμένες μορφές εικόνας)	**Integer image formats (Ακέραιες μορφές εικόνας)**	**Unsigned int image formats (Προσαρμοσμένες μορφές int εικόνας)**
`rgba32f`	`rgba32i`	`rgba32ui`

```
rgba16f            rgba16i            rgba16ui
rg32f              rgba8i             rgb10_a2ui
rg16f              rg32i              rgba8ui
r11f_g11f_b10f     rg16i              rg32ui
r32f               rg8i               rg16ui
r16f               r32i               rg8ui
rgba16             r16i               r32ui
rgb10_a2           r8i                r16ui
rgba8
                                      r8ui
rg16
rg8
r16
r8
rgba16_snorm
rgba8_snorm
rg16_snorm
rg8_snorm
r16_snorm
r8_snorm
```

Όπως έχουμε διαβάσει σε αυτή τη γραμμή στο τελευταίο παράδειγμα, `uniform layout (rgba8) writeonly image2D Image ;`, χρησιμοποιήσαμε τον προσδιοριστή διάταξης(layout) που περικλείει τη μορφή εικόνας. Αυτή η γραμμή καθορίζει επίσης τη λειτουργία πρόσβασης.

Για να δεσμευτεί μια υφή στην εφαρμογή υποδοχής, δεν μπορούμε να χρησιμοποιήσουμε τη συνάρτηση `glBindTexture` όπως συνήθως. Πρέπει να χρησιμοποιήσουμε μια νέα λειτουργία: `glBindImageTexture (0, texID, 0, GL_FALSE, 0, GL_WRITE_ONLY, GL_RGBA8)`. Οι τελευταίες δύο παράμετροι πρέπει να ταιριάζουν με τη δήλωση υφής μέσα στους compute shaders.

Το τελευταίο πράγμα που θα καλύψουμε σχετικά με τους compute shaders που χειρίζονται τις εικόνες υφής είναι ο συγχρονισμός. Η εκτέλεση του compute shader είναι ασύγχρονη, οπότε η λειτουργία `glDispatchCompute` θα επιστρέψει μόλις αποστείλει τις πληροφορίες στη GPU. Δεν θα περιμένει να τελειώσει ο compute shader , οπότε μπορεί να συμβεί αν εκτελέσουμε δύο compute shader που επικαλύπτονται με το χρόνο και ένας από τους shaders γράφει στην εικόνα, ο άλλος shader μπορεί να διαβάσει λανθασμένες τιμές, γι 'αυτό είναι σύνηθες να ισχύει ο shader που γράφει στην εικόνα να τελειώσει πριν εκτελέσει τον άλλο που διαβάζει από την ίδια εικόνα. Αυτό γίνεται με μία από τις λειτουργίες συγχρονισμού που παρέχονται από την OpenGL στο μέρος εφαρμογής του κεντρικού υπολογιστή: `glMemoryBarrier (GL_SHADER_IMAGE_ACCESS_BARRIER_BIT) ;`. Απλά καλούμε αμέσως μετά τη λειτουργία `glDispatchCompute` για να αποκλείσουμε περαιτέρω εκτελέσεις μέχρι να ολοκληρωθεί η εργασία του shader.

8.3 ΥΠΟΛΟΓΙΣΜΟΙ ΑΚΑΤΕΡΓΑΣΤΩΝ ΔΕΔΟΜΕΝΩΝ

Το ακόλουθο δείγμα κώδικα είναι ένα άλλο πολύ απλό παράδειγμα compute shader, αλλά ολοκληρώνει τα βασικά σύνολα λειτουργιών. Πρώτον, χειριστήκαμε τα δεδομένα εικόνας, τώρα, θα χειριστούμε τα ακατέργαστα δεδομένα. Σε αυτό το παράδειγμα, ο shader θα λάβει δύο συστοιχίες του ιδίου μεγέθους και θα χρησιμοποιήσουμε το shader για να τις συγκεντρώσουμε σε μια τρίτη συστοιχία.

Όπως πάντα, ας πάμε πρώτα με τον κώδικα του shader:

```
#version 430
layout (local_size_x = 16, local_size_y = 1, local_size_z = 1) in;
uniform int BufferSize;
layout(std430, binding = 0) buffer InputBufferA{float inA[];};
layout(std430, binding = 1) buffer InputBufferB{float inB[];};
layout(std430, binding=2) buffer OutputBuffer{float outBuffer[];};
void main()
{
uint index = gl_GlobalInvocationID.x;
if(index >= BufferSize)
{
return;
}
outBuffer[index] = inA[index] + inB[index];
}
```

Όπως μπορούμε να δούμε, χρησιμοποιούμε κάθε αντικείμενο εργασίας για να επεξεργαστούμε μόνο ένα στοιχείο μιας συστοιχίας. Τα δύο σημαντικά πράγματα εδώ είναι η δήλωση των buffer και η επιστροφή που ελέγχει την πρόσβαση στους buffer μας.

Το πρώτο πράγμα που πρέπει να παρατηρήσουμε είναι ότι, η δήλωση της συστοιχίας περιέχει ορισμένα στοιχεία. Όπως φαίνεται στα προηγούμενα κεφάλαια, είναι σαν μια δήλωση μπλοκ διασύνδεσης με ορισμένες ιδιαιτερότητες. Ας εξετάσουμε την πρώτη δήλωση buffer:

• Ο όρος `InputBufferA` είναι το όνομα του μπλοκ διεπαφής. Δεν χρησιμοποιείται στον κώδικα.

• Ο `float inA [];` Ο γενικός πίνακας δηλώνει μια σειρά από float στοιχεία και η μεταβλητή που τους συγκρατεί είναι η `inA`.

• Η λέξη-κλειδί του `buffer` δηλώνει τον τύπο του μπλοκ διασύνδεσης.

• Η δήλωση `binding = 0`, δηλώνει ότι αυτή η προσωρινή μνήμη αντιστοιχεί σε εκείνη που συνδέθηκε με το σημείο σύνδεσης 0 στην εφαρμογή υποδοχής.

Η προσδιοριστική λέξη αποθήκευσης `std430` είναι ένας προσδιοριστής διάταξης μνήμης. Περιέχει πληροφορίες σχετικά με την μετατόπιση και την ευθυγράμμιση των δομικών μελών

(αν ο buffer είναι ένας buffer δομών). Προορίζεται να χρησιμοποιηθεί με τα μπλοκ διασύνδεσης shader (εισόδους buffer για compute shader). Αντίθετα, το `std140` πρέπει να χρησιμοποιείται με ομοιόμορφα μπλοκ (ένα ειδικό μπλοκ που συσκευάζει ομοιόμορφες μεταβλητές). Το δεύτερο είναι η κατάσταση που εμποδίζει την πρόσβαση στα όρια των buffer μας. Αυτό θα μπορούσε να συμβεί επειδή μερικές φορές θα μπορούσαμε να έχουμε περισσότερα αντικείμενα εργασίας παγκοσμίως από τα στοιχεία στο buffer μας. Για παράδειγμα, εάν το buffer μας έχει μέγεθος 200 στοιχείων και χρησιμοποιούμε 13 ομάδες εργασίας με 16 αντικείμενα εργασίας ανά ομάδα εργασίας, πραγματοποιεί συνολικά 208 εκτελέσεις και η συστοιχία μας έχει μέγεθος μόνο 200 στοιχείων. Αν δεν ελέγξουμε αυτές τις ξεχωριστές περιπτώσεις, ένα σφάλμα μπορεί να σταματήσει την εκτέλεση του shader, ή ακόμα χειρότερα, να συντρίψει την εφαρμογή.

Όπως και με τις υφές, οι buffer πρέπει να δημιουργούνται και να διαχειρίζονται διαφορετικά για να δουλεύουν με compute shaders (`GL_SHADER_STORAGE_BUFFER`). Ένας νέος τύπος στόχου buffer κορυφής έχει προστεθεί στην OpenGL για να αντικατοπτρίζει αυτό.

Το παρακάτω είναι ένα παράδειγμα εφαρμογής υποδοχής για τον τρόπο εκτέλεσης του προηγουμένως αναφερόμενου compute shader:

```
   constintarraySize = 200;
   float A[arraySize]; // Input array A
   float B[arraySize]; // Input array B
   float O[arraySize]; // Output array
   // fill with some sample values
   for(size_t i = 0; i < arraySize; ++i)
   {
   A[i] = (float)i;
   B[i] = arraySize - i - 1.0f;
O[i] = 10000;
   }
   // Create buffers
   GLuintinA, inB, out;
   glGenBuffers(1, &inA);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, inA); // Bind buffer A
   glBufferData(GL_SHADER_STORAGE_BUFFER, arraySize * sizeof(float),
   A, GL_STATIC_DRAW); // Fill Buffer data
   glGenBuffers(1, &inB);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, inB); // Bind buffer B
   glBufferData(GL_SHADER_STORAGE_BUFFER, arraySize * sizeof(float),
   B, GL_STATIC_DRAW); // Fill Buffer data
   glGenBuffers(1, &out);
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, out); // Bind buffer O
   glBufferData(GL_SHADER_STORAGE_BUFFER, arraySize * sizeof(float),
   O, GL_STATIC_DRAW); // Fill Buffer data
   // Bind buffers to fixed binding points (later will be used in the
   shader)
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 0, inA);
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 1, inB);
   glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 2, out);
```

```
   glUseProgram(computeShaderID); // Bind compute shader
   glDispatchCompute(13, 1, 1); // Execute the compute shader with 13
   workgroups
   glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT   |   GL_SHADER_
   STORAGE_BARRIER_BIT   |   GL_BUFFER_UPDATE_BARRIER_BIT);   //   force
   completeness before read back data
   // Read back the output buffer to check the results
   glBindBuffer(GL_SHADER_STORAGE_BUFFER, out); // Bind output buffer
   // Obtain a pointer to the output buffer data
   float*   data   =   (float*)glMapBuffer(GL_SHADER_STORAGE_BUFFER,
   GL_READ_ ONLY);
   // Copy the data to our CPU located memory buffer
   memcpy(&O[0], data, sizeof(float)*arraySize);
   // Release the GPU pointer
   glUnmapBuffer(GL_SHADER_STORAGE_BUFFER);
// From here, write results to a file, screen or send them back to
memory for further process
```

ΣΥΜΠΕΡΑΣΜΑΤΑ

Στις σελίδες αυτής της εργασίας κάναμε εκτενή περιγραφή των προγραμματιστικών μονάδων και των σχετικών αλγορίθμων που υλοποιούνται σε αυτές για τη φωτορεαλιστική φωτοαπόδοση συνθετικών τρισδιάστατων σκηνικών σε πραγματικό χρόνο. Προχωρήσαμε σε μια ιστορική αναδρομή γύρω από την εξέλιξη των γραφικών τα τελευταία 30 χρόνια και καταλήξαμε βήμα - βήμα στην κατανόηση των OPEN GL και GLSL που αποτελούν και το βασικό μας εργαλείο.

Συμπερασματικά, τα άλματα που έγιναν τα τελευταία χρόνια στο κομμάτι των γραφικών υπολογιστών, συνέβαλαν τα μέγιστα στη βελτίωση της ποιότητας στην εμπειρία της εικόνας, τόσο στον κινηματογράφο όσο και στο gaming αλλά και σε τομείς της επιστήμης και της βιομηχανίας. Ωστόσο, η δομή της προγραμματιζόμενης γραφικής σωλήνωσης έχει βασικό χαρακτηριστικό τη δυνατότητα που δίνει στους χρήστες να εξελίσσουν τις δεξιότητες τους αλλά και να τροποποιούν προς το βέλτιστο την ίδια τη λειτουργία της. Με δεδομένα τα παραπάνω θεωρείται βέβαιο πως τα επόμενα χρόνια η εξέλιξη στο κομμάτι των γραφικών υπολογιστών θα είναι ραγδαία, ανοίγοντας νέους δρόμους για εικονικές εμπειρίες πέρα από τη σφαίρα της φαντασίας.

ΑΝΑΦΟΡΕΣ

John Kessenich, D. R. (7 February 2013). *The OpenGL Shading Language(Language Version 4.30).*

Lengyel, E. (n.d.). *Mathematics for 3D Game Programming and Computer Graphics(2nd Edition).* Charles River Media.

Mark Segal, K. (September 23, 2008). *The OpenGL Graphics System: Aspecification(Version 3.0).*

Mike Bailey, S. (n.d.). *Graphics Shaders: Theory and Practice.*

Rodriguez, J. (χ.χ.). *GLSL Essentials.*

T.Akenine-Möller, E. H. (n.d.). *Real-Time Rendering(3rd Edition).* AK Peters Ltd.