ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΑΤΜΗΜΑΤΙΚΟ ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ
ΣΤΗ ΜΙΚΡΟΗΛΕΚΤΡΟΝΙΚΗ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση σε Τεχνολογία Zynq FPGA της Μονάδας Rasterizer ως Επιταχυντή Υλικού με τη χρήση του Vivado HLS της XILINX

Δημήτριος - Φοίβος Ι. Γερακάρης

Επιβλέπων **Αντώνης Πασχάλης,** Καθηγητής

ΑΘΗΝΑ

Δεκέμβριος 2017

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση σε Τεχνολογία Zynq FPGA της Μονάδας Rasterizer ως Επιταχυντή Υλικού με τη χρήση του Vivado HLS της XILINX

Δημήτριος - Φοίβος Ι. Γερακάρης

Α.Μ.: ΜΜ267

ΕΠΙΒΛΕΠΩΝ: **Αντώνης Πασχάλης,** Καθηγητής

Δεκέμβριος 2017

ΠΕΡΙΛΗΨΗ

Σκοπός της εργασίας αυτής είναι, αρχικά, η αναλυτική περιγραφή των βασικών προβλημάτων και των προτεινόμενων λύσεων στο γνωστικό πεδίο των γραφικών με υπολογιστή και στη συνέχεια η μελέτη της αποτελεσματικότητας του εργαλείου λογισμικού Vivado HLS της XILINX για την ταχεία υλοποίηση σε υψηλό επίπεδο ενός επιταχυντή υλικού σε τεχνολογία Zynq FPGA με μελέτη περίπτωσης τη μονάδα Rasterizer.

Ο στόχος της εργασίας είναι η ανάπτυξη μίας νέας μονάδας Rasterizer σε υψηλό επίπεδο, ως επιταχυντή υλικού, που να δύναται να αντικαταστήσει πλήρως σε λειτουργικότητα την υπάρχουσα μονάδα Rasterizer της εταιρίας Think Silicon που έχει υλοποιηθεί στο επίπεδο RTL με Verilog. Η ανάπτυξη αυτής της εναλλακτικής λύσης προσφέρει στην εταιρία τη δυνατότητα ταχείας δοκιμής εναλλακτικών αλγορίθμων αλλά και τη δυνατότητα γρηγορότερης προσομοίωσης σε επίπεδο λογισμικού.

Τα βήματα που ακολουθήθηκαν ήταν η κατανόηση μέρους του γνωστικού πεδίου των γραφικών με υπολογιστή που αφορούν στη διαδικασία rasterization, η κατανόηση των χρησιμοποιούμενων αλγορίθμων, η κατανόηση του τρόπου που υλοποιούνται αυτοί εντός της GPU της εταιρίας (Nema) και η προσπάθεια να επιτευχθεί η ίδια ή αντίστοιχη λειτουργία μέσω του εργαλείου Vivado HLS. Τέλος έγινε η ενσωμάτωση του κομματιού αυτού στην υπάρχουσα αλυσίδα εργαλείων της εταιρίας, με σκοπό να είναι κρυμμένη η διαδικασία από τον χρήστη.

Από αυτή τη μελέτη περίπτωσης μίας σύνθετης μονάδας, όπως η μονάδα Rasterizer, προκύπτουν πολύ ενδιαφέροντα συμπεράσματα που αφορούν στις προκλήσεις και στην αποτελεσματικότητα της υλοποίησης αλγορίθμων στο υλικό με εργαλεία HLS, όπως το Vivado HLS της XILINX.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Υλικό, Ψηφιακή Επεξεργασία Σήματος, Γραφικά

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: High-Level Synthesis, FPGA, επεξεργαστές υλικού, επεξεργασία εικόνας, γραφικά, rasterization

ABSTRACT

The purpose of this Thesis is to declare and explain the different algorithms around Computer Graphics focusing mainly on rasterization rendering technique and explore the effectiveness of Vivado HLS in rapid graphics rasterization algorithms development. The goal is to be able to produce exact or similar results with company's commercial rasterizer. After the successful implementation, much faster simulation is possible, along with the ability of easy implementation of new algorithms. The whole process designed to be implemented and integrated automatically in the company's toolchain.

SUBJECT AREA: Graphics

KEYWORDS: Image, graphics, rasterizer, Vivado HLS, algorithms

ΕΥΧΑΡΙΣΤΙΕΣ

Στο σημείο αυτό, θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή κ. Αντώνη Πασχάλη για τη στήριξη που μου παρείχε κατά τη διάρκεια αυτής της εργασίας. Επίσης, όλα τους εργαζόμενους και τα στελέχη της εταιρίας Think Silicon για την βοήθειά τους σε όλα τα στάδια της εργασίας καθώς και για την παραχώρηση γραφείου στο παράρτημά τους στην Αθήνα. Τέλος, ένα μεγάλο ευχαριστώ στην οικογένεια μου, η οποία με στηρίζει όλα αυτά τα χρόνια.

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΡΟΛΟΓΟΣ **12**

1. ΕΙΚΟΝΑ **13**

1.1 Εικόνα Raster (bitmap) **13**

1.2 Εικόνα Vector **14**

1.3 Σύγκριση εικόνων Vector και Raster **14**

2. RENDERING **16**

2.1 Rasterization **17**

2.1.1 Κουτί ορίων τριγώνων (2D Triangles Bounding Box): 20

2.1.2 Image or Frame Buffer 21

2.1.3 Αλγόριθμος Βάθους (Depth or Z Buffer) 22

2.1.4 Coverage Test and Edge function 23

2.1.5 Βαρυκεντρικές συντεταγμένες (Barycentric coordinates) 26

2.1.6 Ανάλυση αλγορίθμου Z-Buffer 31

2.1.7 Perspective Correct Interpolation (Αλγόριθμος «σωστής» προοπτικής) 32

2.1.8 Bresenham Algorithm 38

2.2 Ray Casting. **41**

2.3 Ray Tracing. **43**

2.4 Radiosity. **45**

3. FPGA **47**

3.1 Εισαγωγή στα FPGA **47**

3.2 Ιστορική Αναδρομή **47**

3.3 Το Zynq-7000 (ZC702) **49**

3.3.1 Σύστημα Επεξεργασίας (PS) 51

3.3.2 Προγραμματιζόμενη Λογική (PL) 53

3.3.3 Επικοινωνία PS και PL 56

3.4 Προγραμματισμός του FPGA **60**

3.5 Vivado Design Suite (VDS) **61**

3.5.1 Vivado High-Level Synthesis (HLS) 62

3.5.2 Vivado 68

3.5.3 Xilinx Software Development Kit (SDK) 69

3.6 Ενσωματωμένο Linux (Embedded Linux) **70**

4. ΥΛΟΠΟΙΗΣΗ **74**

4.1 Αρχιτεκτονική NemaP GPU **74**

4.2 Περιγραφή αλγορίθμων που υλοποιήθηκαν **75**

4.3 Εξακρίβωση ορθής λειτουργίας του Rasterizer **76**

4.3.1 Line Drawing Bresenham 76

4.3.2 Triangle Drawing 77

4.3.3 Texturing 78

4.3.4 Mixed Shapes 81

4.3.5 Σύγκριση Αποτελεσμάτων 82
4.4 Implementation 84
4.5 Integration 91
5. ΣΥΜΠΕΡΑΣΜΑΤΑ 93
ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ 94
ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ 96
ΑΝΑΦΟΡΕΣ 97

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

Εικόνα 1: Απλή χρωματική παλέτα RGB 13
Εικόνα 2: Μεγέθυνση Raster και Vector εικόνων 15
Εικόνα 3: Vector and Raster 15
Εικόνα 4: Προβολή τριγώνου 18
Εικόνα 5: Τεχνική «γεμίσματος» pixel 18
Εικόνα 6: Όρια τριγώνου 20
Εικόνα 7: Επικαλλυπτόμενα τρίγωνα 22
Εικόνα 8: Ανάλυση Βάθους - Ορατότητας 22
Εικόνα 9: Edge function 24
Εικόνα 10: Edge function στο τρίγωνο 24
Εικόνα 11: Προσδιορισμός κατεύθυνσης με βάση την γωνία 25
Εικόνα 12: Προσδιορισμός σημείου με βάση τις 3 πλευρές του τριγώνου 26
Εικόνα 13: Χρωματισμός τριγώνου με αλγόριθμο βαρυκεντρικών 27
Εικόνα 14: Οπτική αναπαράσταση βαρυκεντρικών 27
Εικόνα 15: Οπτική βαρυκεντρικών με λ0=0 28
Εικόνα 16: Μεταβολή αναλογίας εμβαδών 28
Εικόνα 17: Ανάλυση επιμέρους εμβαδών -- βαρυκεντρικών 29
Εικόνα 18: Interpolate - Extrapolate 30
Εικόνα 19: Πρόβλημα κατανομής βάση βάθους 31
Εικόνα 20: Σφάλμα αναλογίας αποστάσεων 31
Εικόνα 21: Κάθετη οπτική τετραγώνου 33
Εικόνα 22: Πρόβλημα προοπτικής τετραγώνου 33
Εικόνα 23: Υπολογισμός Z 34
Εικόνα 24: Σύγκριση barycentric - perspective correct interpolation 37
Εικόνα 25: Επιβεβαίωση ορθής λειτουργίας του αλγορίθμου μέσω οπτικής σύγκρισης 37
Εικόνα 26: Διαφορά barycentric - perspective correct στο chess matrix 38
Εικόνα 27: Λειτουργία Bresenham και υπολογισμός λαθών 39
Εικόνα 28: Απεικόνιση του τρόπου λειτουργίας της μεθόδου Ray Casting 42
Εικόνα 29: Απεικόνιση ενός τρισδιάστατου αντικειμένου στην δισδιάστατη οθόνη του υπολογιστή 42
Εικόνα 30: Απεικόνιση του τρόπου λειτουργίας της μεθόδου Ray Tracing 44
Εικόνα 31: Φωτορεαλιστική εικόνα μέσω Ray Casting 44
Εικόνα 32: Οπτικά αποτελέσματα για διαφορετικό αριθμό βημάτων Radiosity 45
Εικόνα 33: Απεικόνιση της λειτουργίας της μεθόδου Radiosity και βελτίωσης της σκίασης από α) εφαρμογή απευθείας φωτισμού σε β) με την εφαρμογή της μεθόδου radiosity . 46
Εικόνα 34: Αρχιτεκτονική Zynq 49

Εικόνα 35: Abstract αρχιτεκτονική Zynq 50
Εικόνα 36: Αναλυτική δισύνδεση στο Zynq 51
Εικόνα 37: APU Block Diagram 51
Εικόνα 38: Zynq I/O Interfaces 52
Εικόνα 39: Progammable Logic του Zynq - CLB 53
Εικόνα 40: Σύνθεση ενός CLB 54
Εικόνα 41: Block Ram και DSP στην PI του Zynq 55
Εικόνα 42: Αρχιτεκτονική DSP48E1 55
Εικόνα 43: Interfaces PS/PI 57
Εικόνα 44: Αφαιρετικό μοντέλο προγραμματισμού Zynq 60
Εικόνα 45: Vivado Design Suite Design Flow 61
Εικόνα 46: Μοντέλο Παραγωγής RTL μέσω Vivado HLS 62
Εικόνα 47: Fixed point number 64
Εικόνα 48: Vivado HLS design flow 68
Εικόνα 49: Vivado Design Flow 68
Εικόνα 50: Vivado Block Diagram 69
Εικόνα 51: Βασικά στοιχεία πυρήνα Linux 71
Εικόνα 52: Αντιστοίχιση Linux Boot Stages - Hardware 72
Εικόνα 53: Επιμέρους στοιχεία BOOT.BIN 72
Εικόνα 54: Συγκεντρωτικά αρχεία για εκκίνηση Zynq 73
Εικόνα 55: Αρχιτεκτονική Nema GPU 74
Εικόνα 56: Πλάγιες γραμμές σχεδιασμένες με τον Bresenham algorithm 76
Εικόνα 57: Οριζόντιες και κάθετες γραμμές σχεδιασμένες με τον Bresenham algorithm 76
Εικόνα 58: Ολοκληρωμένη εικόνα σχεδιασμού τριγώνων 77
Εικόνα 59: Μεγένθυνση τριγώνου για εξακρίβωση λειτουργίας αλγορίθμου 78
Εικόνα 60: Συνδιασμός Γραμμών, rectangles και texturing 78
Εικόνα 61: Χρωματισμός της εικόνας της Σαντορίνης με τη χρήση texturing 79
Εικόνα 62: Εφαρμογή texture διαφανοποίησης στην εικόνα της Σαντορίνης 80
Εικόνα 63: Συνδυασμός όλων των σχημάτων και χρωματισμός με γραμμικό αλγόριθμο 81
Εικόνα 64: Matmult Resources without Pipelining 84
Εικόνα 65: HLS time scheduling 85
Εικόνα 66: HLS Matmult Resources and timing 86
Εικόνα 67: HLS Pipelined time scheduling 86
Εικόνα 68: Questa protected warnings 88
Εικόνα 69: Modelsim encrypted waveform signals 89

Εικόνα 70: Xilinx encrypted netlist 89

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

Πίνακας 1: Διεπαφές μεταξύ PS και PI 58
Πίνακας 2: Data types στην C 63
Πίνακας 3: Arbitary types στο Vivado HLS 63
Πίνακας 4: Fixed point numbers στο HLS 64
Πίνακας 5: Πρωτόκολλα διεπεφής στο HLS 65
Πίνακας 6: Στάδια εκκίνησης Linux 72
Πίνακας 7: Πόροι Rasterizer σε Verilog 82
Πίνακας 8: Πόροι Rasterizer σε HLS 82
Πίνακας 9: Συγκεντρωτικός πίνακας πόρων Rasterizer 82

ΠΡΟΛΟΓΟΣ

Η παρούσα εργασία πραγματοποιήθηκε στα πλαίσια διπλωματικής εργασίας του Διατμηματικού Προγράμματος Μεταπτυχιακών Σπουδών στη Μικροηλεκτρονική, το οποίο λαμβάνει χώρα στο Τμήμα Πληροφορικής και Τηλεπικοινωνιών του Εθνικού και Καποδιστριακού Πανεπιστημίου Αθηνών (ΕΚΠΑ). Η συγκεκριμένη εργασία πραγματοποιήθηκε στις εγκαταστάσεις της εταιρίας Think Silicon.

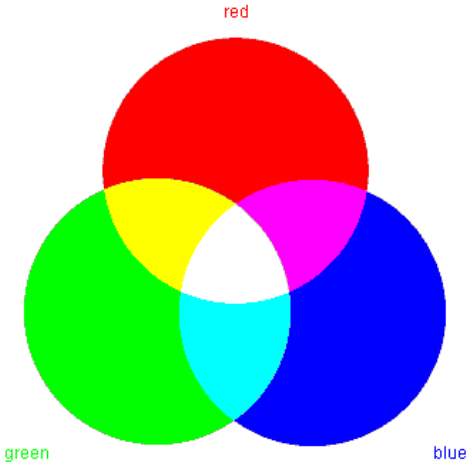
1. ΕΙΚΟΝΑ

Η εικόνα, είναι η αποτύπωση ενός αντικειμένου, μία δυσδιάστατης ή τρισδιάστατης σκηνής του χώρου σε κάποιο μέσο όπως το χαρτί, ο υπολογιστής ή ο ανθρώπινος νους. Μία εικόνα μπορεί να θεωρηθεί ως μία συνεχής συνάρτηση, η οποία σε κάθε σημείο της εμφανίζει κάποιο χρώμα. Όπως είναι λογικό, για να την εισάγουμε-επεξεργαστούμε στον υπολογιστή θα πρέπει με κάποιον τρόπο να τη ψηφιοποιήσουμε. Όπως και με τον ήχο, όσο περισσότερα δυαδικά ψηφία (bits) μπορούμε να δαπανήσουμε για την κάθε εικόνα, τόσο λιγότερη απώλεια πληροφορίας θα έχουμε. Σχετικά με τον υπολογιστή υπάρχουν δύο βασικές κατηγορίες εικόνων: οι εικόνες raster (bitmap) και οι εικόνες vector.

1.1 Εικόνα Raster (bitmap)

Αποτελεί το βασικό είδος εικόνας υπολογιστή ειδικά όταν έχουμε να κάνουμε με φωτογραφίες. Κάθε εικόνα αποτελείται από επιμέρους στοιχεία τα οποία ονομάζουμε εικονοστοιχεία (pixels). Κάθε ένα από αυτά τα pixels, αποτελεί σημείο της διακριτής συνάρτησης, που δημιουργήσαμε από τη συνεχή συνάρτηση του πραγματικού κόσμου. Κάθε pixel έχει μια τιμή η οποία αντικατοπτρίζει το χρώμα της καθώς και ενδεχομένως κάποιες άλλες ιδιότητες (π.χ. διαφάνεια).

Όπως αναφέρθηκε, κατά τη ψηφιοποίηση, θα υπάρχει κάποια απώλεια πληροφορίας. Ο αριθμός των pixels που αναφέραμε καθορίζει την ανάλυσή (resolution) της και ο αριθμός των bits πληροφορίας του καθενός pixel, καθορίζει το βάθος χρώματος (color depth). Κάθε χρώμα αποτελείται από μία ποσότητα κόκκινου (Red), πράσινου (Green) και μπλε (Blue). Ένα απλό χρωματικό RGB μοντέλο παρουσιάζεται στην παρακάτω εικόνα:



Εικόνα 1: Απλή χρωματική παλέτα RGB

Τα δυο ακραία χρώματα του μοντέλου είναι το μαύρο (παντελής έλλειψη χρώματος) και το άσπρο (ίση ποσότητα και από τα τρία χρώματα).

Το ανθρώπινο μάτι μπορεί να ξεχωρίσει περίπου 10 εκατομμύρια χρώματα. Για να μπορέσουμε να καλύψουμε αυτό το εύρος χρωμάτων, συνήθως για ρεαλιστικές εικόνες χρησιμοποιούμε 8 bit ανά χρώμα δηλαδή σύνολο 24 bit. Με αυτόν τον τρόπο είμαστε ικανοί να αναπαραστήσουμε λίγο περισσότερα από 16 εκατομμύρια χρώματα (16.777.216). Αν προσθέσουμε σε αυτά και 8 bit της διαφάνειας κάθε χρώματος, τότε χρειαζόμαστε 32 bit για κάθε pixel. Το πρότυπο με τα 24 bit αποκαλείται RGB χώρος χρωμάτων (color space) και αναφέρεται συχνά ως πραγματικό χρώμα (true color), ενώ το πρότυπο με τα 32 bit αποκαλείται RGBA color space. Σε περίπτωση που μας

ενδιαφέρουν ασπρόμαυρες εικόνες, τότε χρειαζόμαστε απλά 1 bit ανά pixel. Δεδομένων των παραπάνω συνοπτικά μπορούμε να πούμε ότι:

Μια εικόνα raster είναι ένας δισδιάστατος πίνακας I(x,y) όπου ως x αναφερόμαστε στην στήλη της εικόνας και ως y στη γραμμή της. Αν είναι στο 32bit RGBA τότε αφιερώνουμε 1 byte (8 bit) σε κάθε χρώμα καθώς και 8 bit στην διαφάνειά του.

Τα πιο γνωστά πρότυπα εικόνων raster είναι: jpg, gif, png, tif, bmp.

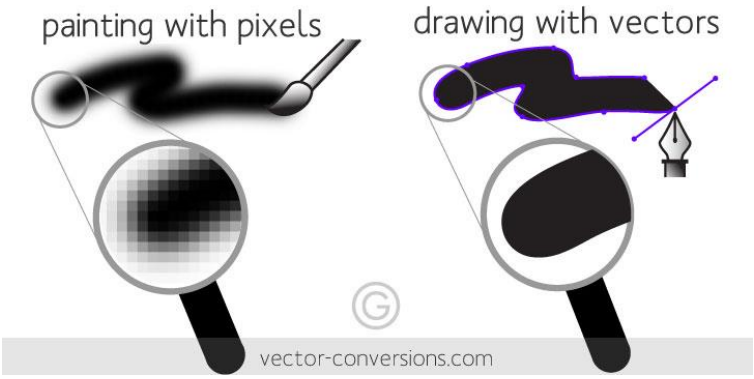
1.2 Εικόνα Vector

Οι vector (διανυσματικές) εικόνες αποτελούνται από λεπτές γραμμές (thin lines) και καμπύλες (curves) που ονομάζονται και μονοπάτια (paths) και έχουν τη βάση τους στη μαθηματική γεωμετρία. Κατασκευάζονται με ειδικό λογισμικό στον υπολογιστή και χρησιμοποιούνται κυρίως για την κατασκευή γραμματοσειρών (fonts) και λογοτύπων (logos) [1]. Καθότι περιγράφουν διανύσματα, έχουν το μεγάλο πλεονέκτημα ότι σε οποιαδήποτε μεγέθυνση παράγουν ακριβώς το ίδιο αποτέλεσμα χωρίς να υπάρχει το πρόβλημα της απώλειας ποιότητας και pixelαρισματος καθώς και το πλεονέκτημα ότι καταλαμβάνουν σημαντικά μικρότερο χώρο από αντίστοιχες raster εικόνες. Τα πιο γνωστά πρότυπα vector αρχείων είναι: ai, cdr, svg.

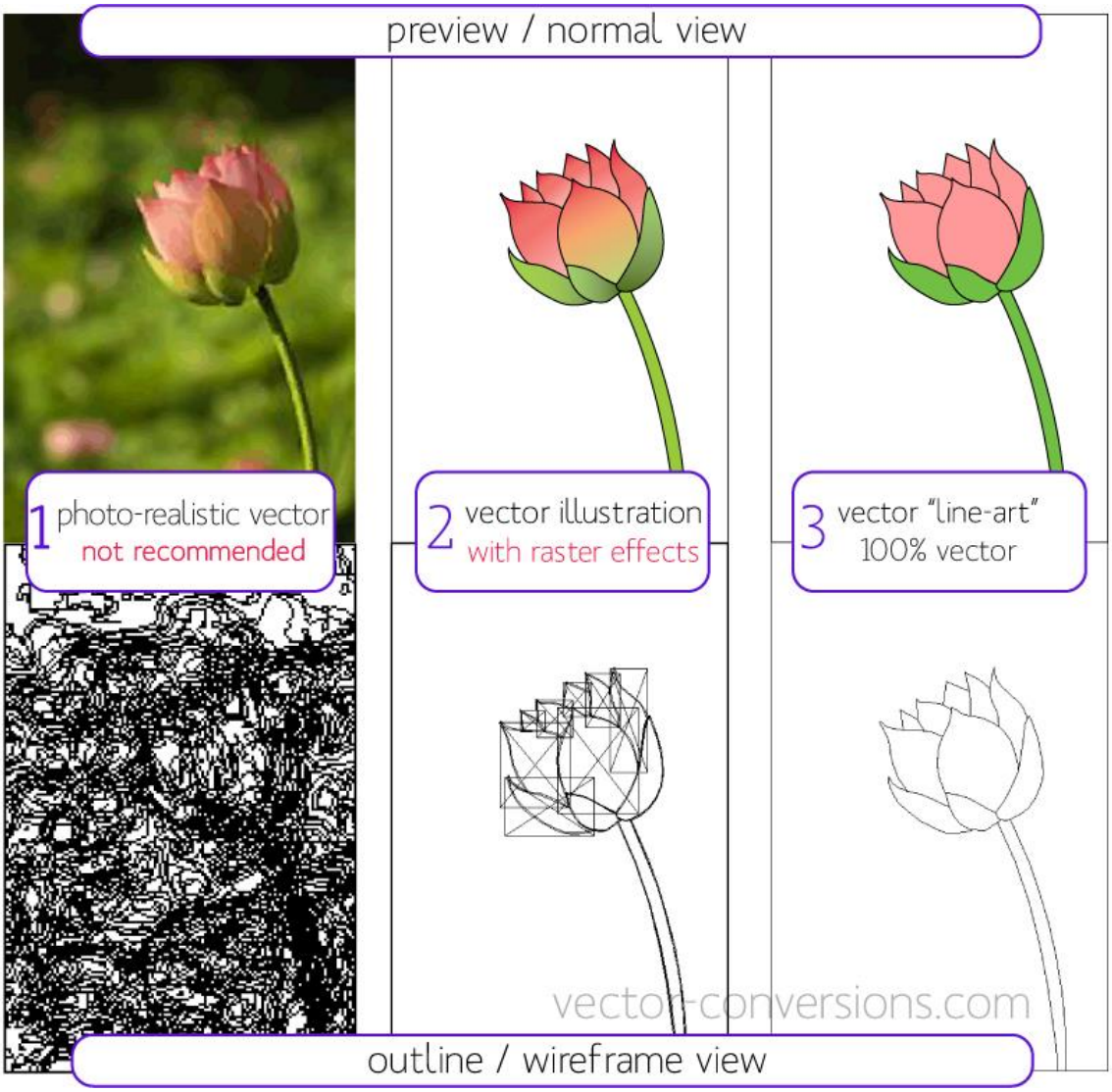
1.3 Σύγκριση εικόνων Vector και Raster

Vector	Raster
Μαθηματικοί υπολογισμοί που σχηματίζουν σχήματα	Αποτελούνται από pixels
Χρησιμοποιούνται κυρίως σε λογότυπα, γραμματοσειρές και σχέδια	Χρησιμοποιούνται σε φωτορεαλιστικές φωτογραφίες και για την κατασκευή γραφικών με ομαλή αλλαγή και μίξη χρωμάτων (blend)
Μπορούν να μεγεθυνθούν χωρίς να αλλάξει η ποιότητά τους	Σε περίπτωση που θέλουμε να τις μεγεθύνουμε χωρίς απώλεια ποιότητας, πρέπει να την κατασκευάσουμε εξ' αρχής ή να τη σκανάρουμε σε μεγαλύτερη ανάλυση
Δεν αλλάζει ο όγκος του αρχείου κατά τη μεγένθυνση	Αύξηση του μεγέθους συνεπάγεται με αύξηση του όγκου που καταναλώνει
Μετατρέπεται εύκολα σε raster	Η μετατροπή σε vector μπορεί να είναι χρονοβόρα και με κακό αποτέλεσμα διαδικασία

Και οι δυο τύποι εικόνας μπορούν να συμμετέχουν στη δημιουργία εικόνας όπως για παράδειγμα κείμενο με vectors και εικόνα με raster ή εξωτερικό σχέδιο με vector και χρώμα με raster effects.



Εικόνα 2: Μεγέθυνση Raster και Vector εικόνων



Εικόνα 3: Vector and Raster

2. RENDERING

Το rendering μεταφράζεται ως απόδοση ή φωτοαπόδοση και είναι η διαδικασία δημιουργίας μίας εικόνας από ένα μοντέλο (2D ή 3D) ή μία ολόκληρη σκηνή, με τη βοήθεια ηλεκτρονικού υπολογιστή. Μία σκηνή (scene) περιέχει αντικείμενα σε μία αυστηρά καθορισμένη γλώσσα ή δομή δεδομένων. Περιλαμβάνει τη γεωμετρία (geometry), τις υφές (texture), τον φωτισμό (lighting) και την σκίαση (shading), την οπτική γωνία και προοπτική της κάμερας (viewpoint), όλα αυτά σαν πληροφορίες για την περιγραφή της εικονικής σκηνής. Τα δεδομένα της σκηνής στη συνέχεια υφίστανται μεταποίηση από το πρόγραμμα (renderer) με στόχο την παραγωγή μίας εικόνας. Αν και οι τεχνικές λεπτομέρειες διαφέρουν ανάμεσα στις μηχανές rendering, η γενική ιδέα για την παραγωγή μια δισδιάστατης εικόνας (2D) από μία τρισδιάστατη αναπαράσταση (3D) είναι ίδια, η λύση της εξίσωσης rendering. Η εξίσωση της απόδοσης (rendering equation) δεν λαμβάνει υπόψη όλα τα φαινόμενα που σχετίζονται με το φως αλλά είναι ένα γενικό μοντέλο φωτισμού για την δημιουργία εικόνας [2].

Το rendering είναι ένα από τα σημαντικότερα θέματα των 3D γραφικών. Είναι το τελευταίο στάδιο σε μια φωτορεαλιστική διαδικασία, δίνοντας την τελική εμφάνιση των μοντέλων. Με την αυξανόμενη πολυπλοκότητα των γραφικών του υπολογιστή από το 1970 και μετά, το rendering έχει γίνει ένα ξεχωριστό θέμα στην τεχνολογία των γραφικών.

Ουσιαστικά ο renderer είναι ένα προσεκτικά κατασκευασμένο πρόγραμμα, βασισμένο σε πολλούς επιστημονικούς κλάδους όπως η φυσική, η οπτική, τα μαθηματικά και η ανάπτυξη λογισμικού. Χωρίζονται σε δύο είδη, αυτό του πραγματικού χρόνου (real time rendering) και αυτό που επιλέγεται από τον χρήστη πότε και σε πόσο χρόνο θα αποδοθεί (pre-rendering). Ένα πρόγραμμα real time rendering χρησιμοποιείται κυρίως σε 3D παιχνίδια, ενώ pre-rendering για την κατασκευή φωτορεαλιστικών εικόνων και βίντεο. Η πρώτη διαδικασία χρησιμοποιεί πολλά καρέ εικόνων που πρέπει να προστεθούν και να ταξινομηθούν σωστά σε ένα πρόγραμμα ικανό να δημιουργήσει animation. Ενώ στη δεύτερη διαδικασία ένα πλέγμα (wireframe) επεξεργάζεται και προθέτονται πάνω του υφές, φώτα και άλλα για την δημιουργία της ζητούμενης εικόνας.

Κατά τη διάρκεια της ανάπτυξης της επιστήμης των υπολογιστών αλλά και της διαθεσιμότητας ολοένα και υψηλότερης απόδοσης μηχανημάτων έχουν αναπτυχθεί αρκετές διαδικασίες rendering διαφορετικών χαρακτηριστικών και πολυπλοκότητας. Η επιλογή τους γίνεται ανάλογα το αποτέλεσμα που θέλουμε να επιτύχουμε και της επεξεργαστικής ισχύος και χρόνου που διαθέτουμε. Οι διαδικασίες με βάση τη διαδρομή του φωτός ταξινομούνται σε 3 (4) βασικές οικογένειες:

1. Rasterization
2. Ray casting
3. Ray tracing
4. Radiosity (χρησιμοποιείται κυρίως επικουρικά)

Συνήθως τα περισσότερο εξελιγμένα προγράμματα χρησιμοποιούν τουλάχιστον δυο διαδικασίες ώστε να επιτύχουν το καλύτερο αποτέλεσμα με το γρηγορότερο δυνατό τρόπο [3]. όλες οι αναφερθείσες διαδικασίες χρησιμοποιούν τρόπους για να μειωθούν τα σωματίδια του φωτός που «παρακολουθούνται» για το σχηματισμό της εικόνας. Η «παρακολούθηση» κάθε σωματιδίου κάθε πηγής φωτός είναι απλά αδύνατη ακόμα και με τους σημερινούς υπολογιστές. Μια διαφορετική προσέγγιση διαχωρισμού διαδικασιών είναι μεταξύ αυτών που επεξεργάζονται τα pixel της εικόνας (image order) και εκείνων που επεξεργάζονται τα αντικείμενα της εικόνας (object order). Στη συνέχεια αναλύουμε τις προαναφερθείσες διαδικασίες.

2.1 Rasterization

Rasterisation (και Rasterization) είναι η διαδικασία κατά την οποία μια εικόνα vector μετατρέπεται σε εικόνα raster, ώστε να σχεδιαστεί σε pixel είτε στην οθόνη μας είτε σε κάποιον εκτυπωτή είτε σε κάποιο raster πρότυπο (format) αρχείο εικόνας όπως πχ το bmp ή το jpg. Το rasterization αφορά είτε 3D μοντέλα είτε 2D αρχέτυπα (primitives) όπως είναι τα πολύγωνα, οι γραμμές κλπ. Αυτήν την στιγμή, η συγκεκριμένη μέθοδος είναι η πιο κοινή στην περίπτωση που θέλουμε να αναπαραστήσουμε γραφικά πραγματικού χρόνου (real-time computer graphics), καθότι σε εφαρμογές όπως πχ τα παιχνίδια απαιτούνται τουλάχιστον 30 fps (frames per second) για να έχουμε ομαλή κίνηση. Ο αλγόριθμος του rasterization προτιμάται σε σχέση με «ανταγωνιστικούς» όπως πχ το ray casting καθώς είναι γρηγορότερος, προσφέροντας όμως λιγότερο ρεαλιστικά γραφικά.

Ιστορικά οι περισσότεροι αλγόριθμοι του rasterization έχουν αναπτυχθεί μεταξύ του 1960 και του 1980. Προφανώς το υλικό έχει αλλάξει από τότε, αλλά οι αλγόριθμοι παραμένουν ίδιοι. Και ο ray-tracing αλγόριθμος και ο ray casting προσπαθούν να επιλύσουν το πρόβλημα της ορατότητας (visibility) ή αλλιώς της κρυμμένης επιφάνειας (hidden surface) [4] δηλαδή συνοπτικά του προβλήματος όταν δυο αντικείμενα στο 3D περιβάλλον εμφανίζονται το ένα πιο μπροστά στην κάμερα από το άλλο, άρα και μόνο το ένα εξ αυτών θα αποτυπωθεί στην οθόνη καθώς και το πρόβλημα της σκίασης (shading). Και οι δύο χρησιμοποιούν τεχνικές της υπολογιστικής γεωμετρίας (CG) στην υλοποίηση και εμφανίζουν όμοια αποτελέσματα τουλάχιστον στον τομέα του visibility. Η επίλυση του προβλήματος αυτού μπορεί να λυθεί ουσιαστικά με δυο τρόπους (ray tracing και rasterization). Ο πρώτος είναι να ακολουθήσουμε κάθε ακτίνα που διαπερνάει κάθε pixel και να βρούμε τα σημεία τομής αυτής της ακτίνας με κάθε αντικείμενο της σκηνής που τέμνει. Μετά συγκρίνουμε το σημείο τομής (intersection point) καθενός σχήματος και επιλέγουμε να εμφανιστεί αυτό που έχει την μικρότερη απόσταση από την κάμερα. Αυτός είναι και ο τρόπος που ακολουθείται στη μέθοδο ray tracing.

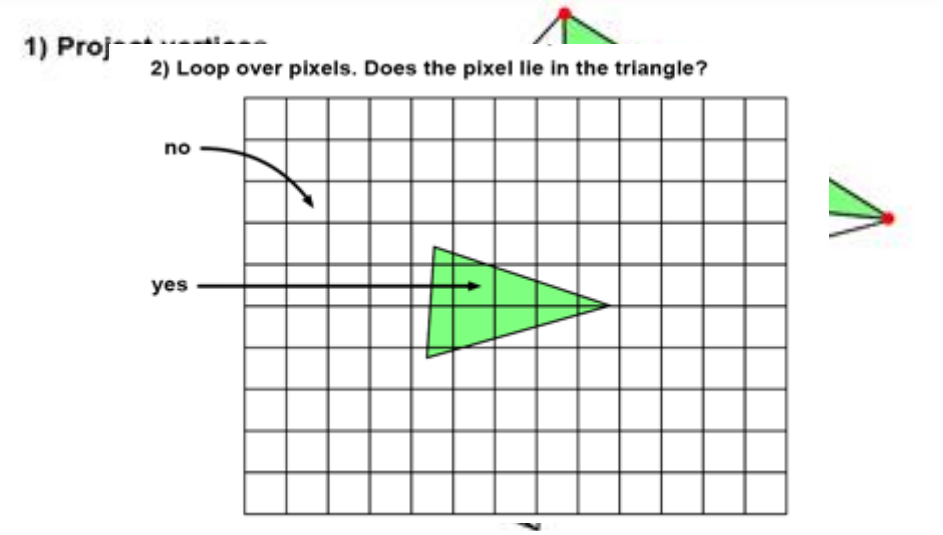
Ο συγκεκριμένος αλγόριθμος επί της ουσίας χρειάζεται δυο επαναλήψεις (loops). Μια εξωτερική για κάθε pixel και μια εσωτερική που εξετάζει κατά πόσο τέμνει το κάθε σχήμα της σκηνής.

```
for (each pixel in image) {
    Ray R = computeRayPassingThroughPixel(x,y);
    float tclosest = INFINITY;
    Triangle triangleClosest = NULL;
    for (each triangle in scene) {
        float thit;
        if (intersect(R, object, thit)) {
            if (thit < closest) {
                triangleClosest = triangle;
            }
        }
```

Όπως παρατηρούμε τα αντικείμενα της σκηνής θεωρούνται ότι αποτελούνται από τρίγωνα και μόνο. Τα τρίγωνα είναι το βασικότερο αρχέτυπο (primitive) των γραφικών υπολογιστή και στις δυο μεθόδους και είναι αυτό που χρησιμοποιείται κατά κόρον από τις GPUs. Σε αντίθεση με τον ray tracing που είναι image centric αλγόριθμος, καθότι ακολουθεί ακτίνες από την κάμερα προς τη σκηνή, ο rasterization ακολουθεί την ανάποδη

διαδρομή. Δηλαδή «προβάλλει» το τρίγωνο στην οθόνη μετατρέποντας το 3D σχήμα σε 2D ακολουθώντας τον αλγόριθμο προοπτικής (perspective projection).

Στη συνέχεια ακολουθείται μία τεχνική «γεμίσματος» ή χρωματίσματος των pixel που το προβληθέν τρίγωνο καταλαμβάνει στην οθόνη.



Εικόνα 5: Τεχνική «γεμίσματος» pixel

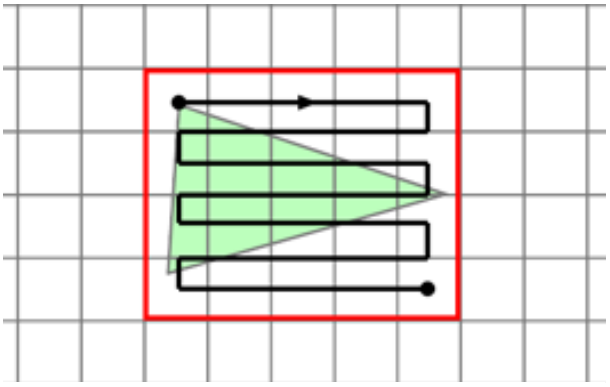
```
// rasterization algorithm
for (each triangle in scene) {
   // STEP 1: project vertices of the triangle using perspective projection
   Vec2f v0 = perspectiveProject(triangle[i].v0);
   Vec2f v1 = perspectiveProject(triangle[i].v1);
   Vec2f v2 = perspectiveProject(triangle[i].v2);
   for (each pixel in image) {
      // STEP 2: is this pixel contained in the projected image of the triangle?
      if (pixelContainedIn2DTriangle(v0, v1, v2, x, y)) {
         image(x,y) = triangle[i].color;
      }
   }
}
```

Από τεχνικής άποψης ο αλγόριθμος προβολής χρειάζεται μόνο μια διαίρεση της απόστασης και έναν επαναϋπολογισμό των συντεταγμένων από τον χώρο της σκηνής στο χώρο της εικόνας αλλά και ο υπολογισμός αν κάποιο Pixel είναι μέρος της εικόνας είναι απλός υπολογισμός. Σε σχέση με τον ray tracing εδώ οι loops είναι ανεστραμμένες. Δηλαδή αντί να ακολουθούμαι κάθε Pixel της εικόνας ακολουθούμε κάθε τρίγωνο της σκηνής και εσωτερικά εξετάζουμε αν κάθε pixel της εικόνας αποτελεί μέρος του συγκεκριμένου τριγώνου. Δηλαδή :

Ο συγκεκριμένος αλγόριθμος είναι object centric αφού ξεκινάμε από το αντικείμενο και πηγαίνουμε προς την σκηνή, προσέγγιση αντίθετη του ray tracing. Επί της αρχής και οι δυο αλγόριθμοι είναι απλοί στην σκέψη, αλλά ειδικά ο ray tracing μπορεί να αποβεί σχετικά δύσκολος στον υπολογισμό της διατομής καθώς και αρκετά ακριβός επεξεργαστικά. Στον rasterizer, όπως αναφέραμε ο υπολογισμός και των 2 διαδικασιών που απαιτεί είναι αρκετά απλός και εύκολα παραλληλοποιήσιμος εξ' ου και χρησιμοποιείται στις GPUs. Εκτός αυτών η απόδοσή του είναι εύκολα προβλέψιμη. Για αυτό και είναι ο πιο συνήθης στην παραγωγή 3D αντικειμένων. Παρόλο που είναι απλός και εύκολα παραλληλοποιήσιμος, χρησιμοποιούμε διάφορες τεχνικές ώστε να μειώσουμε τις περιττές ενέργειες και να επιταχύνουμε τον αλγόριθμο ώστε να μπορούμε να επεξεργαστούμε όσο το δυνατόν περισσότερα στοιχεία.

2.1.1 Κουτί ορίων τριγώνων (2D Triangles Bounding Box):

Στην προαναφερθείσα περίπτωση του αλγορίθμου, εξετάζαμε όλα τα pixel της εικόνας αν ανήκουν στο τρίγωνο. Αυτή η αφελής προσέγγιση θα μας κόστιζε πολλούς κύκλους ερευνώντας (πολλά) κενά pixels. Για να το αποφύγουμε αυτό υπολογίζουμε τα άκρα των συντεταγμένων του τριγώνου και εξετάζουμε τα pixel μόνο στο εσωτερικό αυτού του



Εικόνα 6: Όρια τριγώνου

διαστήματος. Δεδομένου ότι κάθε αντικείμενο αποτελείται από εκατοντάδες έως εκατομμύρια τρίγωνα, αντιλαμβανόμαστε πόσο χρόνο μας εξοικονομεί αυτή η απλή απλούστευση. Προφανώς ακόμα και σε αυτήν την περίπτωση δεν εξαλείφουμε τον έλεγχο των κενών pixels αλλά ανάλογα την γεωμετρία του τριγώνου τα περιορίζουμε σε έναν μεγάλο βαθμό.

Ο υπολογισμός των συντεταγμένων του κουτιού γίνεται προφανώς στο επίπεδο raster και μπορεί εύκολα να περιγραφεί από τον ακόλουθο αλγόριθμο:

```
// convert the vertices of the current triangle to raster space
Vec2f bbmin = INFINITY, bbmax = -INFINITY;
Vec2f vproj[3];
for (int i = 0; i < 3; ++i) {
    vproj[i] = projectAndConvertToNDC(triangle[i].v[i]);
    // coordinates are in raster space but still floats not integers
    vproj[i].x *= imageWidth;
    vproj[i].y *= imageHeight;
    if (vproj[i].x < bbmin.x) bbmin.x = vproj[i].x);
    if (vproj[i].y < bbmin.y) bbmin.y = vproj[i].y);
    if (vproj[i].x > bbmax.x) bbmax.x = vproj[i].x);
    if (vproj[i].y > bbmax.y) bbmax.y = vproj[i].y);
}
```

Μόλις υπολογίσουμε τα όρια διατρέχουμε κάθε pixel του κουτιού. Η μόνη προσοχή στην συγκεκριμένη περίπτωση είναι σε δυο σημεία:

1. Είναι πιθανόν κατά την προβολή του τριγώνου στο επίπεδο raster, μέρος του τριγώνου να είναι εκτός σκηνής. Άρα το κομμάτι αυτό δεν θα πρέπει να ελεγχθεί

εφόσον δεν θα προβληθεί και θα ήταν σπατάλη πόρων. Άρα θα πρέπει τα όρια του κουτιού να μην ξεπερνούν αυτά της σκηνής.

2. Κατά την προβολή του τριγώνου στη σκηνή συνήθως χρησιμοποιούμε float συντεταγμένες ενώ τα pixel συνήθως είναι integer αριθμοί. Άρα θα πρέπει να κάνουμε στρογγυλοποίηση στον κοντινότερο integer αριθμό. Έτσι ο αλγόριθμος παίρνει την παρακάτω μορφή:

```
uint xmin = std::max(0, std:min(imageWidth - 1, std::floor(min.x)));
uint ymin = std::max(0, std:min(imageHeight - 1, std::floor(min.y)));
uint xmax = std::max(0, std:min(imageWidth - 1, std::floor(max.x)));
uint ymax = std::max(0, std:min(imageHeight - 1, std::floor(max.y)));
for (y = ymin; y <= ymin; ++y) {
   for (x = xmin; x <= xmax; ++x) {
      // check of if current pixel lies in triangle
      if (pixelContainedIn2DTriangle(v0, v1, v2, x, y)) {
         image(x,y) = triangle[i].color;
      } } }
```

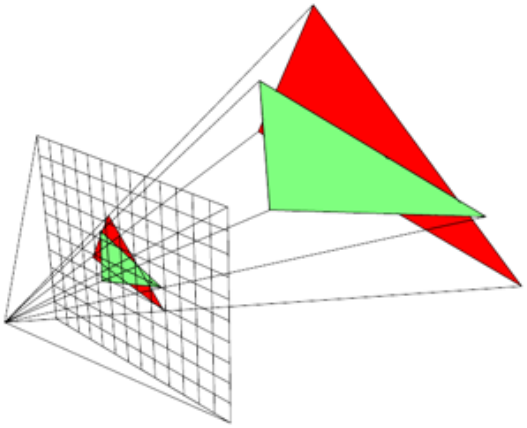
Υπάρχουν και πιο σύνθετοι αλγόριθμοι ώστε να εξετάσουμε αν κάποιο pixel ενδέχεται να ανήκει στο τρίγωνο ή όχι. Επίσης πρέπει να αναφερθεί πολλές φορές κυρίως παλιότερα οι συντεταγμένες των διανυσμάτων μετατρέπονταν σε fixed point numbers από floating points καθώς οι πράξεις μεταξύ fixed points οι οποίοι επί της ουσίας είναι πράξεις μεταξύ ακεραίων είναι αρκετά γρηγορότερες από τις αντίστοιχες των floating point. Βέβαια πλέον σχεδόν όλες οι gpu διαθέτουν floating point μονάδες οπότε και είναι παραπλήσιος ο χρόνος εκτέλεσης και άρα δεν χρησιμοποιείται η μετατροπή.

2.1.2 Image or Frame Buffer

Πριν το ξεκίνημα του rendering η εικόνα που είναι έτοιμη να αποτυπωθεί στην οθόνη είναι ένα σύνολο από κενά (μαύρα pixels). Δηλαδή ένας πίνακας από X*Y στοιχεία όπου X το πλάτος και Y το ύψος της εικόνας. Κατά τη διάρκεια επεξεργασίας των τριγώνων της σκηνής, το κάθε pixel της εικόνας αποκτά κάποιο χρώμα (το οποίο μπορεί να αλλάξει αρκετές φορές). Η τελική τιμή του κάθε στοιχείου της εικόνας θα καθορίσει και το αποτέλεσμα που θα εμφανιστεί. Αυτός ο «πίνακας στοιχείων» ονομάζεται image ή frame buffer και μπορεί είτε να αποτυπωθεί στην οθόνη είτε να αποθηκευτεί σε κάποιο αρχείο ώστε να το επεξεργαστούμε μετά με κάποιο συμβατό πρόγραμμα.

2.1.3 Αλγόριθμος Βάθους (Depth or Z Buffer)

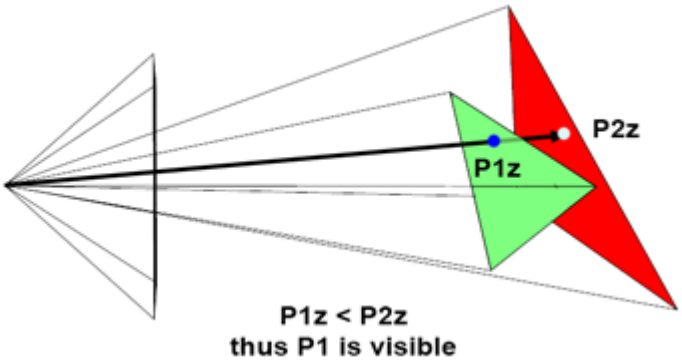
Όπως αναφέραμε προηγουμένως ο βασικός στόχος του rasterization είναι να επιλύσει το πρόβλημα της ορατότητας (visibility). Ιστορικά δυο αλγόριθμοι χρησιμοποιήθηκαν αρχικά για την επίλυση του προβλήματος: Ο αλγόριθμου του Newell και ο Z-buffer. Ο τελευταίος είναι αυτός που χρησιμοποιείται σήμερα στις gpu. Για να έχουμε ένα λειτουργικό βασικό rasterizer πρέπει να αντιμετωπίσουμε το γεγονός ότι σε μία σκηνή δυο οι περισσότερα τρίγωνα μπορεί να συμπέφτουν στο ίδιο σημείο. Άρα πρέπει να βρούμε έναν τρόπο ώστε να αποφασίσουμε ποιο από τα δυο θα συμβάλλει στην τελική εικόνα. Για αυτόν τον λόγο εκτός από τον δισδιάστατο πίνακα για τα χρώματα, χρησιμοποιούμε ακόμα έναν που αυτήν την φορά αρχικοποιείται αντί στο 0, σε έναν πολύ μεγάλο αριθμό. Όταν ένα σημείο που εξετάζουμε ανήκει στο τρίγωνο, τότε εξετάζουμε και την αντίστοιχη τιμή του βάθους, δηλαδή της απόστασης αυτού του σημείου από την κάμερα. Στην αρχή όπως αναφέραμε αυτός θα είναι ένας πολύ μεγάλος αριθμός (ενδεχομένως το άπειρο). Αν το βάθος του συγκεκριμένου τριγώνου είναι μικρότερο από το βάθος του υπάρχοντος pixel, τότε αντικαθιστούμε το χρώμα που υπάρχει στην εικόνα με το καινούριο που προκύπτει από



Εικόνα 7: Επικαλλυπτόμενα τρίγωνα

το τρίγωνο που επεξεργαζόμαστε και φυσικά αντικαθιστούμε το βάθος του παλιού τριγώνου με το νέο. Προφανώς αν η απόσταση που ήδη έχει ο z buffer είναι μικρότερη από την τωρινή, δεν αλλάζουμε ούτε τον frame buffer ούτε τον z-buffer.

Όπως μπορούμε να διακρίνουμε και στην διπλανή εικόνα το πράσινο τρίγωνο είναι πιο κοντά από το κόκκινο οπότε και ο z-buffer θα δώσει μικρότερη τιμή στο $P1_z$ και άρα η τελική εικόνα θα είναι όπως στην εικόνα που ακολουθεί:



Εικόνα 8: Ανάλυση Βάθους - Ορατότητας

Ένας απλός αλγόριθμος που θέτει την σωστή τιμή στον z-buffer είναι ο ακόλουθος:

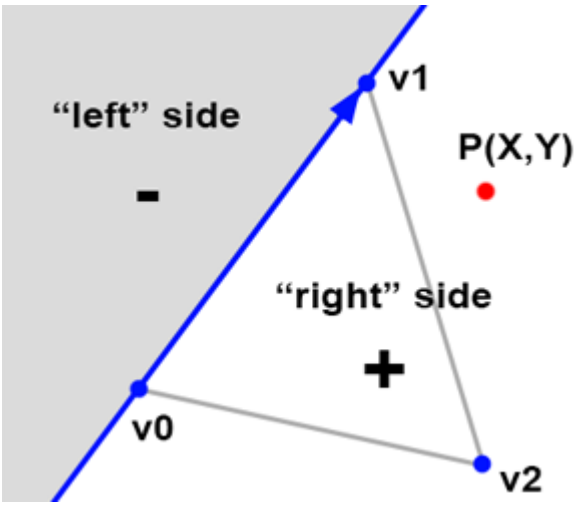
```
// A z-buffer is just an 2D array of floats
float buffer = new float [imageWidth * imageHeight];
// initialize the distance for each pixel to a very large number
for (uint32_t i = 0; i < imageWidth * imageHeight; ++i)
buffer[i] = INFINITY;

for (each triangle in scene) {
   // project vertices
   ...
   // compute bbox of the projected triangle
   ...
   for (y = ymin; y <= ymin; ++y) {
      for (x = xmin; x <= xmax; ++x) {
         // check of if current pixel lies in triangle
         float z; // distance from the camera to the triangle
         if (pixelContainedIn2DTriangle(v0, v1, v2, x, y, z)) {
            // If the distance to that triangle is lower than the distance stored in the
            // z-buffer, update the z-buffer and update the image at pixel location (x,y)
            // with the color of that triangle
            if (z < zbuffer(x,y)) {
               zbuffer(x,y) = z;
               image(x,y) = triangle[i].color; } } } } }
```

Για τον συγκεκριμένο κώδικα θεωρούμε ότι η συνάρτηση pixelContainedIn2DTriangle υπολογίζει την απόσταση του pixel από την κάμερα.

2.1.4 Coverage Test and Edge function

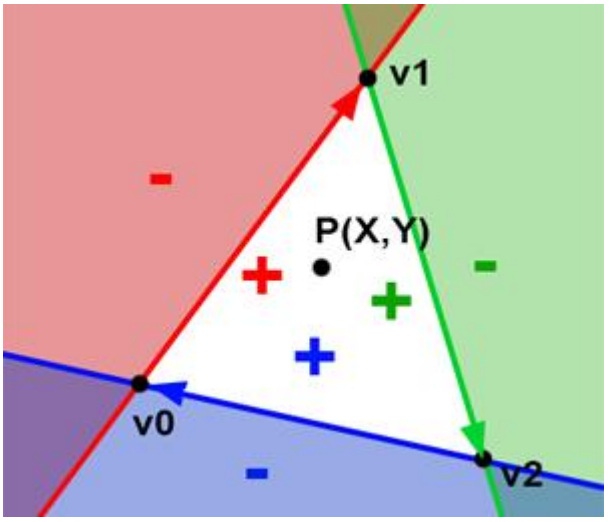
Προηγουμένως, αναφέραμε ότι ένα βασικό κομμάτι του rasterization είναι να καθορίσουμε ποια σημεία ακριβώς ανήκουν στο τρίγωνο. Η διαδικασία αυτή στα APIs των GPUs ονομάζεται inside-outside test ή πιο απλά coverage test. Η τεχνική που χρησιμοποιείται για να προσδιοριστεί η θέση του σημείου ονομάζεται edge function. Η μέθοδος αυτή παρουσιάστηκε στο paper [5] του Juan Pineda το 1988 με τίτλο “A Parallel Algorithm for Polygon Rasterization”. Η βασική αρχή της μεθόδου του Pineda είναι η εύρεση μιας μαθηματικής συνάρτησης την οποία αποκαλούμε edge function (συνάρτηση ακμής) κατά την οποία ένα σημείο που ανήκει αριστερά της ακμής αυτής επιστρέφει αρνητική τιμή στη συνάρτηση ενώ ένα σημείο δεξιά της ακμής επιστρέφει θετική τιμή. Εάν η τιμή που θα επιστραφεί είναι μηδενική, αυτό σημαίνει ότι το σημείο είναι πάνω στην ευθεία της ακμής.

Στην ακόλουθη εικόνα μπορούμε να δούμε μια σχηματική αναπαράσταση της edge function



Εικόνα 9: Edge function

Όπως μπορούμε να διακρίνουμε το διάνυσμα v0-v1 έχει συγκεκριμένη φορά. Αυτό είναι πολύ σημαντικό για την εφαρμογή της μεθόδου. Εφαρμόζοντας λοιπόν την συγκεκριμένη μέθοδο σε κάθε διάνυσμα (v0-v1, v1-v2, v2-v0) μπορούμε να προσδιορίσουμε αν το σημείο (έστω P) που εξετάζουμε ανήκει στο τρίγωνο. Αν όλες οι εξισώσεις επιστρέψουν θετικό αποτέλεσμα, προφανώς το σημείο ανήκει εντός του τριγώνου.



Εικόνα 10: Edge function στο τρίγωνο

Εφόσον αναφέραμε την γενική ιδέα της συνάρτησης θα προχωρήσουμε σε μια πιο λεπτομερή ανάλυση. Για το διάνυσμα v0-v1 (δηλαδή την αριστερή πλευρά του τριγώνου) ισχύει η εξής σχέση:

$E01(P) = (P.x - v0.x) * (V1.y - V0.y) - (P.y - V0.y) * (V1.x - V0.x)$. Στη σχέση όπως είναι φυσικό περιγράφεται η σχέση μεταξύ του σημείου P και του διανύσματος v0-v1. Αν:

E(P) > 0 τότε το P ανήκει στη δεξιά μεριά

E(P) < 0 τότε το P ανήκει στην αριστερή μεριά

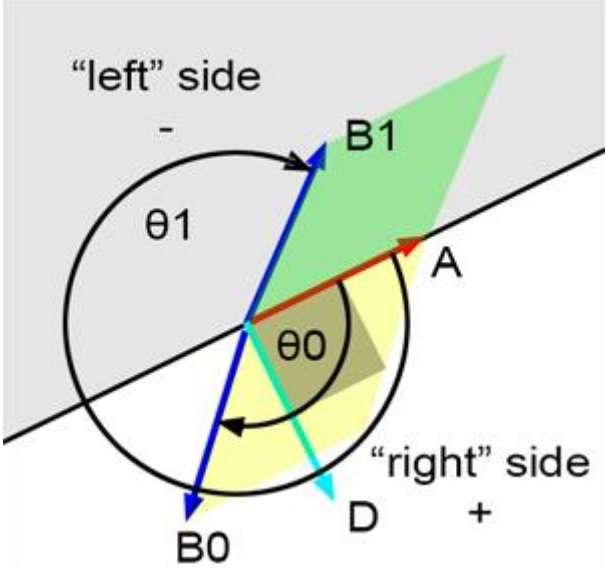
E(P) = 0 τότε το P ανήκει πάνω στη γραμμή που ορίζει το διάνυσμα v0-v1

Η πιο πάνω σχέση, μπορεί να γραφεί και ως η διακρίνουσα ενός 2x2 πίνακα όπου το πρώτο διάνυσμα A = (P-V0) και B = (V1-V0). Έτσι μπορούμε να γράψουμε τη συνάρτηση ακμής:

$$\begin{bmatrix} A.x & A.y \\ B.x & B.y \end{bmatrix} = A.x * B.y - A.y * B.x = (P.x - V0.x) * (V1.y - V0.y) - (P.y - V0.y) * (V1.x - V0.x)$$

Αν παρατηρήσουμε καλύτερα την εξίσωση, θα αντιληφθούμε ότι πρόκειται για το μέτρο του εξωτερικού γινομένου των δυο αναφερθέντων διανυσμάτων A και B. Όπως γνωρίζουμε η επιφάνεια μεταξύ δυο διανυσμάτων, υπολογίζεται από το εξωτερικό γινόμενο από τον τύπο

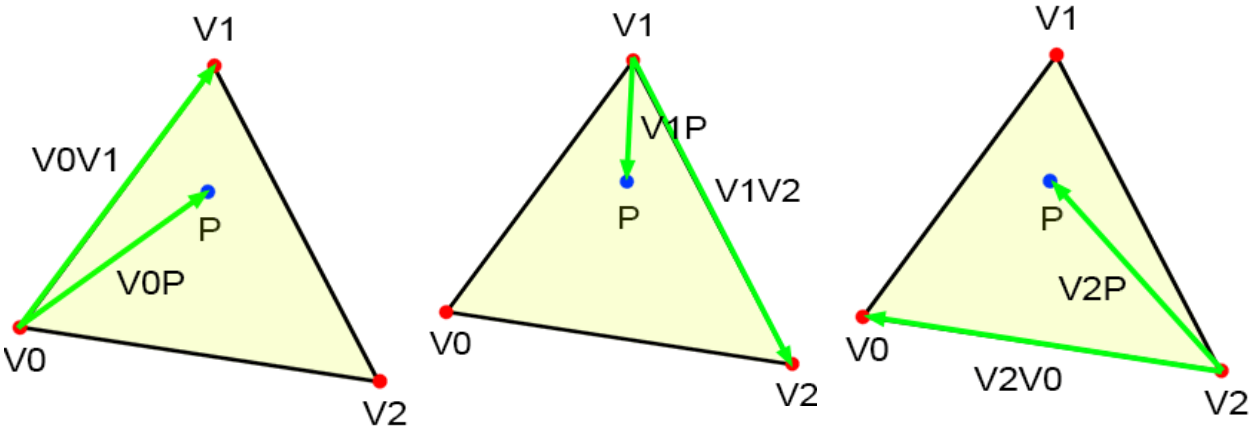
$Area = ||AxB|| = ||A||||B|| \sin(\theta)$. Προφανώς η επιφάνεια δεν μπορεί να είναι αρνητική, άρα μπορούμε από το πρόσημό της να αντιληφθούμε μια σχέση ανάμεσα στην κατεύθυνση των A και B. Στο επόμενο σχήμα έχουμε μια εικόνα που θα μας διευκολύνει.



Εικόνα 11: Προσδιορισμός κατεύθυνσης με βάση την γωνία

Όπως παρατηρούμε στην περίπτωση που η γωνία μεταξύ του A και του B είναι μικρότερη των 180° τότε το αποτέλεσμα της εξίσωσης είναι θετικό γιατί το B βρίσκεται στην «δεξιά» μεριά. Αν είναι η γωνία είναι μεγαλύτερη τότε το B σημαίνει ότι βρίσκεται στην αριστερή - αρνητική πλευρά. Δηλαδή αν το θ ανήκει στο διάστημα (0,π) τότε το αποτέλεσμα είναι θετικό, ενώ αν ανήκει στο (π,2π) είναι αρνητικό. Στις περιπτώσεις 0,π είναι 0.

Εφόσον λοιπόν ξεκαθαρίσαμε τη λογική κατά την οποία «καταλαβαίνουμε» αν ένα σημείο είναι στην θετική ή την αρνητική μεριά μιας ευθείας, επεκτείνουμε τον έλεγχο και στις 3 ευθείες που καθορίζονται από το τρίγωνο



Εικόνα 12: Προσδιορισμός σημείου με βάση τις 3 πλευρές του τριγώνου

Επιλέγουμε τα διανύσματα ακολουθώντας τη φορά του ρολογιού (δηλαδή δεξιόστροφα). Για τον καθορισμό αυστηρά της απόφασης για τη θέση του σημείου P δεν ενδιαφερόμαστε για το «μέτρο» της συνάρτησης απλά για το πρόσημό της. Αναφορικά και μόνο, το μέτρο μας είναι χρήσιμο για τον προσδιορισμό των βαρυκεντρικών συντεταγμένων (barycentric coordinates) τα οποία ενδεχομένως θα μας χρειαστούν στο interpolation.

Για ολόκληρο το τρίγωνο λοιπόν οι εξισώσεις θέσης ορίζονται από τους τύπους:

$$E01(P) = (P.x - V0.x) * (V1.y - V0.y) - (P.y - V0.y) * (V1.x - V0.x),$$
$$E12(P) = (P.x - V1.x) * (V2.y - V1.y) - (P.y - V1.y) * (V2.x - V1.x),$$
$$E20(P) = (P.x - V2.x) * (V0.y - V2.y) - (P.y - V2.y) * (V0.x - V2.x).$$

Έτσι αν όλες οι εξισώσεις είναι μη αρνητικές, τότε το σημείο ανήκει στο τρίγωνο (στην περίπτωση 0 ανήκει σε μία πλευρά του τριγώνου). Προφανώς υπάρχουν και άλλες μέθοδοι για τον υπολογισμό αν ένα σημείο ανήκει σε ένα τρίγωνο, η ανάλυση των οποίων ξεπερνά τα πλαίσια της διπλωματικής. Αναφορικά, ο πιο γνωστός είναι ο standard αλγόριθμος scanline, ο Bresenham αλγόριθμος, ο βαρυκεντρικός κ.ο.κ. Αυτός όμως που αναλύσαμε είναι ο πλέον διαδεδομένος στις κάρτες γραφικών λόγω της απλότητας αλλά και της παραλληλοποίησής του.

2.1.5 Βαρυκεντρικές συντεταγμένες (Barycentric coordinates)

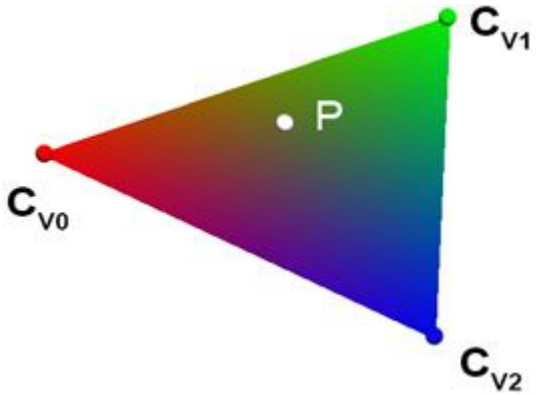
Οι βαρυκεντρικές συντεταγμένες χρησιμοποιούνται για τον προσδιορισμό διαφόρων ιδιοτήτων (attributes) των διανυσμάτων. Μπορεί να αφορούν είτε το χρώμα, είτε την υφή, είτε την νόρμα κλπ ανάλογα με τον αλγόριθμο που θέλουμε να χρησιμοποιήσουμε. Αρχικά, για τα τρίγωνα, είναι σετ 3 αριθμών κινητής υποδιαστολής (floating point (fp)).

Έστω λ0, λ1, λ2. Για τους αριθμούς αυτούς ισχύουν τα εξής:

$P = \lambda 0 * V0 + \lambda 1 * V1 + \lambda 2 * V2$ (1) όπου P είναι το σημείο του τριγώνου και V0,V1,V2 τα διανύσματα που αναφέραμε και στα προηγούμενα παραδείγματα.

$\lambda 0 + \lambda 1 + \lambda 2 = 1, for\ P \in \triangle V0, V1, V2$ (2). Πολλές φορές αναφέρονται και ως βάρη (weights) των κορυφών του τριγώνου. Μπορούμε να φανταστούμε ότι κάθε σημείο του τριγώνου «περιέχει» ένα κομμάτι κάποιας ιδιότητας της κορυφής V1 της V2 και V3. Αν για παράδειγμα λ1=1 αυτό σημαίνει ότι το σημείο P είναι η κορυφή V1.

Παράδειγμα: Έστω τρίγωνο όπου CV0 = κόκκινο, Cv1=πράσινο και Cv2= μπλε. Ποιο θα ήταν το χρώμα κάθε σημείου του τριγώνου με βάση τον αλγόριθμο των βαρυκεντρικών συντεταγμένων?



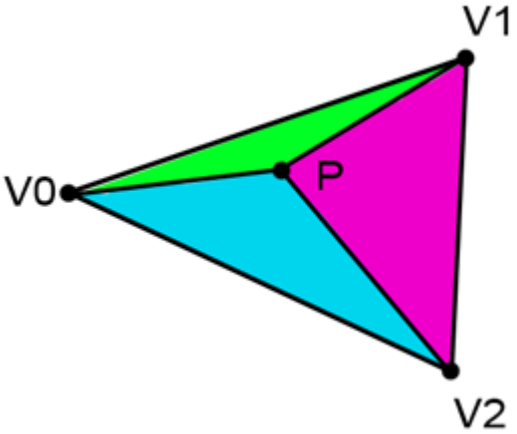
Εικόνα 13: Χρωματισμός τριγώνου με αλγόριθμο βαρυκεντρικών

Η τιμή του κάθε σημείου δίνεται από τον τύπο:

$$CP = \lambda 0 * CV0 + \lambda 1 * CV1 + \lambda 2 * CV2$$

Το καλό με τον συγκεκριμένο αλγόριθμο είναι ότι μπορεί να χρησιμοποιηθεί για διάφορες ιδιότητες των κορυφών όχι μόνο για το χρώμα (όπως πχ η υφή).

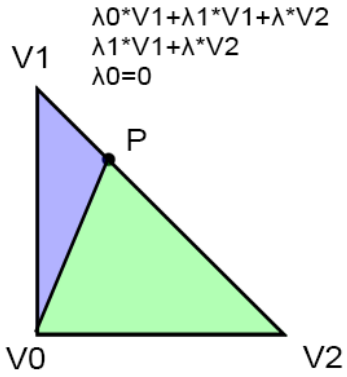
Πώς όμως υπολογίζουμε τις βαρυκεντρικές συντεταγμένες? Έστω ότι έχουμε το παρακάτω τρίγωνο:



Εικόνα 14: Οπτική αναπαράσταση βαρυκεντρικών

Όπως παρατηρούμε το σημείο P ορίζει 3 υποτρίγωνα,. ένα ροζ, ένα πράσινο και ένα γαλάζιο. Η συνολική επιφάνεια δίνεται από τον τύπο $\mathrm{Area}_{\triangle V0V1V2} = \mathrm{Area}_{\triangle V0V1P} + \mathrm{Area}_{\triangle V1V2P} + \mathrm{Area}_{\triangle V2V0P}$.

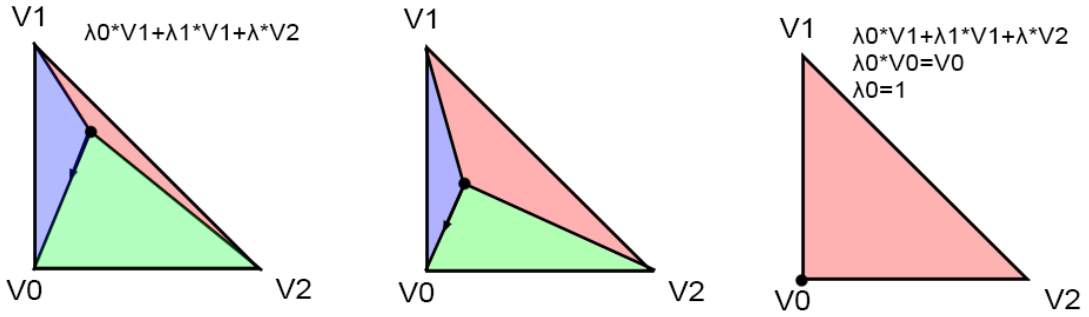
Για να καταλάβουμε καλύτερα τον αλγόριθμο καλύτερα να ακολουθήσουμε τμηματικά τι συμβαίνει:



Εικόνα 15: Οπτική βαρυκεντρικών με λ0=0

Στην αρχική αυτή περίπτωση, παρατηρούμε ότι δεν έχουμε καθόλου ροζ τρίγωνο καθότι το σημείο P βρίσκεται πάνω στην πλευρά V2-V1. Άρα το λ0 = 0 στην περίπτωση αυτή. Η εξίσωση δηλαδή διαμορφώνεται σε $P = \lambda 1 * V1 + \lambda 2 * V2$. Εφόσον $\lambda 1 + \lambda 2\ (\lambda 0 = 0) = 1$ καταλαβαίνουμε ότι $\lambda 2 = 1 - \lambda 1$. Επίσης παρατηρούμε ότι το σημείο P είναι πιο κοντά στο V1 από ότι στο V2 και αυτό μας προϊδεάζει στο γεγονός ότι το λ1 είναι μεγαλύτερο από το λ2. Επίσης παρατηρούμε ότι η επιφάνεια του μωβ τριγώνου είναι σαφώς μεγαλύτερη της πράσινης. Παρατηρούμε δηλαδή μια σχέση μεταξύ της τιμής των βαρυκεντρικών συντεταγμένων με τις επιφάνειες των τριγώνων.

Τώρα μετακινώντας σταδιακά το σημείο P προς την κορυφή V0 παρατηρούμε την αλλαγή της αναλογίας των επιφανειών των τριγώνων.

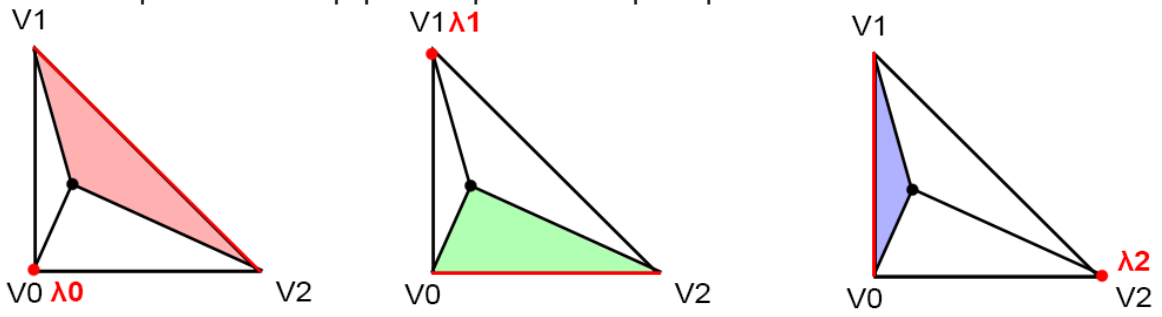


Εικόνα 16: Μεταβολή αναλογίας εμβαδών

Το λ0 είναι ανάλογο με την επιφάνεια του ροζ τριγώνου, το λ1 με την επιφάνεια του πράσινου και το λ2 με την επιφάνεια του μωβ.

Στο τελευταίο τρίγωνο παρατηρούμε μόνο ροζ καθώς το λ0=1 αφού συμπίπτει με την V0 κορυφή και λ1,λ2 = 0 εφόσον $\lambda 0 + \lambda 1 + \lambda 2 = 1$.

Μια καλύτερη εικόνα σχετικά με τη συσχέτιση μεταξύ των βαρυκεντρικών συντεταγμένων και των επιφανειών των τριγώνων φαίνεται στην παρακάτω εικόνα:



Εικόνα 17: Ανάλυση επιμέρους εμβαδών -- βαρυκεντρικών

Μια σημαντική παρατήρηση: Στην προηγούμενη ενότητα αναφέραμε εκτενώς την edge function και είπαμε ότι το μέτρο της δεν ήταν ιδιαίτερα χρήσιμο για την εξακρίβωση θέσης, αλλά θα ήταν χρήσιμο για τις βαρυκεντρικές συντεταγμένες. Το εξωτερικό γινόμενο που υπολογίζαμε στη συγκεκριμένη συνάρτηση αφορούσε το παραλληλόγραμο που καθόριζε το εκάστοτε ζεύγος διανυσμάτων. Όπως είναι γνωστό το τρίγωνο που καθορίζουν τα δυο αυτά διανύσματα είναι το μισό σε επιφάνεια από το παραλληλόγραμμο. Έτσι οι για τις αντίστοιχες εξισώσεις διαμορφώνονται ως εξής:

$$Area_{tri}(V1, V2, P) = \left(\frac{1}{2}\right) * E_{12}(P)$$

$$Area_{tri}(V2, V0, P) = \left(\frac{1}{2}\right) * E_{20}(P)$$

$$Area_{tri}(V0, V1, P) = \left(\frac{1}{2}\right) * E_{01}(P).$$

Όπως παρατηρήσαμε και προηγουμένως οι τιμές των λ συσχετίζονταν με την αναλογία των εμβαδών των τριγώνων που σχημάτιζαν με το σημείο P και του συνολικού τριγώνου. Οι τιμές των των λ δίνονται από τις σχέσεις:

$$\lambda 0 = \frac{Area(V1, V2, P)}{Area(V0, V1, V2)}$$

$$\lambda_1 = \frac{Area(V2, V0, P)}{Area(V0, V1, V2)}$$

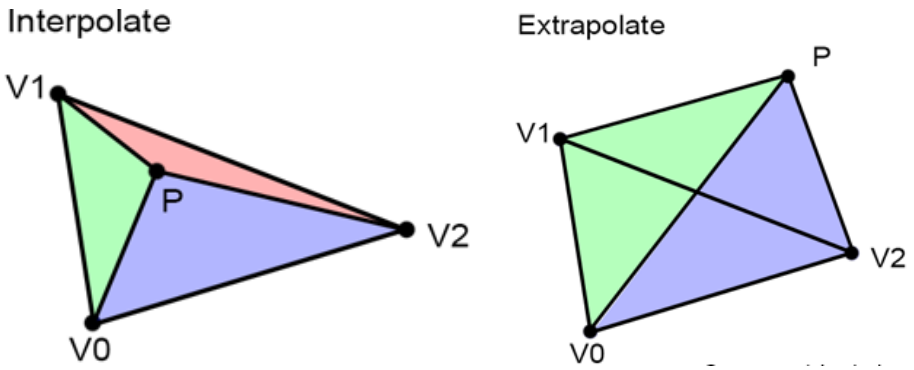
$$\lambda_2 = \frac{Area(V0, V1, P)}{Area(V0, V1, V2)}$$

Η διαίρεση με το εμβαδόν του τριγώνου γίνεται με σκοπό να «νορμαλοποιηθεί» το αποτέλεσμα.

Δεδομένων των παραπάνω, μπορούμε εύκολα να τροποποιήσουμε τον αλγόριθμο της προηγούμενης ενότητας, αντί να μας καθορίζει μόνο αν ένα pixel βρίσκεται εντός του τριγώνου, να υπολογίζει και τις βαρυκεντρικές του συντεταγμένες. Ο αλγόριθμος διαμορφώνεται ως εξής:

```
float edgeFunction(const Vec2f &a, const Vec3f &b, const Vec2f &c)
{
    return (c.x - a.x) * (b.y - a.y) - (c.y - a.y) * (b.x - a.x);
}

float area = edgeFunction(v0, v1, v2); // area of the triangle multiplied by 2
float w0 = edgeFunction(v1, v2, p); // signed area of the triangle v1v2p multiplied by 2
float w1 = edgeFunction(v2, v0, p); // signed area of the triangle v2v0p multiplied by 2
float w2 = edgeFunction(v0, v1, p); // signed area of the triangle v0v1p multiplied by 2

// if point p is inside triangles defined by vertices v0, v1, v2
if (w0 >= 0 && w1 >= 0 && w2 >= 0) {
    // barycentric coordinates are the areas of the sub-triangles divided by the area of
the main triangle
    w0 /= area;
    w1 /= area;
    w2 /= area;
}
```

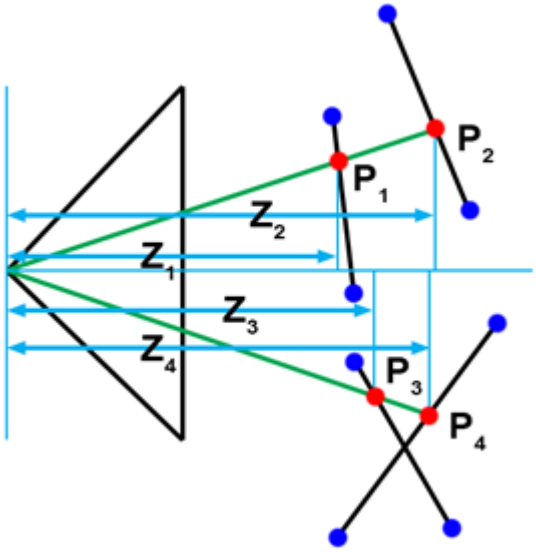
Αξίζει να σημειωθεί ότι ο ανωτέρω αλγόριθμος μπορεί να χρησιμοποιηθεί και για τον υπολογισμό του Z (θεωρητικά) κάθε σημείου του τριγώνου. Με αυτόν τον τρόπο με μόνο δυο προσθέσεις και πολλαπλασιασμούς υπολογίζουμε και την απόσταση Z. Τέλος, ο αλγόριθμος όπως παρατηρούμε (Εικ, 18) παράγει αποτελέσματα και για σημεία εκτός τριγώνου. Σε κάποιες περιπτώσεις όπως ενδεχομένως των υφών, είναι χρήσιμοι οι υπολογισμοί αυτοί. Αν το σημείο είναι εντός του τριγώνου τότε κάνουμε interpolate ενώ εκτός extrapolate.



Εικόνα 18: Interpolate - Extrapolate

2.1.6 Ανάλυση αλγορίθμου Z-Buffer

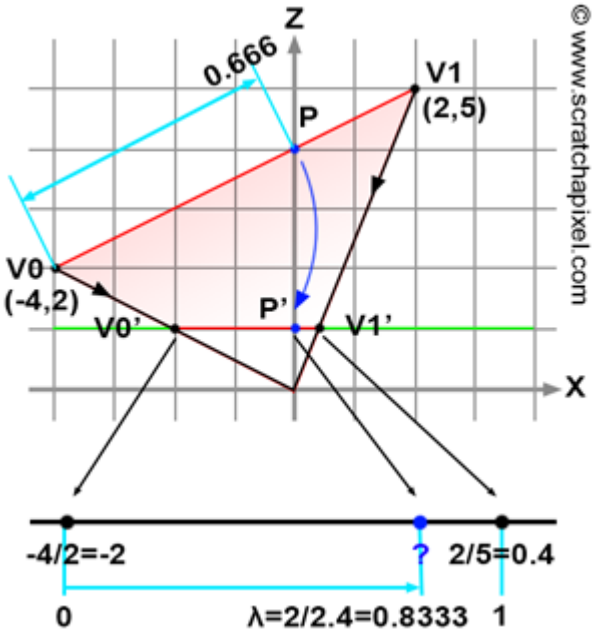
Όπως αναφέραμε και σε προηγούμενη ενότητα βασικό πρόβλημα επίλυσης του rasterizer είναι ο προσδιορισμός των ορατών τριγώνων στην οθόνη, σε περίπτωση που 2 ή περισσότερα από αυτά «συμπίπτουν». Μια αφελής σχετικά λύση θα ήταν να ταξινομήσουμε κατά σειρά απόστασης τα τρίγωνα και να υπολογίζαμε την απόσταση μόνο του κοντινότερου. Αυτό θα ήταν σωστό στην περίπτωση των σημείων P1 και P2 του διπλανού σχήματος. Τι γίνεται όμως στην περίπτωση P3 και P4 δηλαδή σε αυτά που κάποιο τμήμα του τριγώνου είναι εν μέρη πίσω από ένα άλλο και κάποιο μπροστά; Τότε πολύ απλά θα εμφανίσουμε στην οθόνη «λάθος» pixels. Το πρόβλημα αυτό ερχόμαστε να λύσουμε με τον Z-Buffer αλγόριθμο. Όπως εξηγήσαμε, για κάθε σημείο που εξετάζουμε ως προς τη θέση του σε σχέση με το τρίγωνο, υπολογίζουμε και την τιμή του Z για το σημείο αυτό. Πως όμως υπολογίζουμε την τιμή αυτή;



Εικόνα 19: Πρόβλημα κατανομής βάση βάθους

Με βαρυκεντρικές συντεταγμένες είναι η προφανής απάντηση (αφού είναι ήδη υπολογισμένες με την edge function).

Δυστυχώς αυτός ο τρόπος δεν λειτουργεί αξιόπιστα. Ας δούμε ένα παράδειγμα.



Εικόνα 20: Σφάλμα αναλογίας αποστάσεων

Έστω ότι θέλουμε να προβάλουμε το σημείο P (0,4) στο επίπεδο της πράσινης γραμμής (z=1). Το σημείο τομής είναι το P' με συντεταγμένες (0,1). Το αντίστοιχο για το V0 (-4,2) είναι το V0' (-2,1). Αντίστοιχα για το V1 (2,5) το V1' (0,4,1).

Γενικά οι τιμές υπολογίζονται από τον τύπο V0'.x= V0.x/V0.z. Αν πάμε να υπολογίσουμε τις βαρυκεντρικές συντεταγμένες έχουμε:

$$\lambda = \frac{[P'.x - V0'.x)}{V1'.x - V0'.x} = \frac{(0 - -2)}{(0.4 - -2)} = \frac{2}{2.4} = 0.833$$

Αν αντικαταστήσουμε στη σχέση των βαρυκεντρικών έχουμε:

$$P.z = V0.z * (1 - \lambda) + V1.z * \lambda = 2 * 1.666 + 5 * 0.833 = 4.5.$$

Όπως βλέπουμε αυτό δεν είναι το αναμενόμενο αποτέλεσμα το οποίο είναι P.z = 4. Αυτό συμβαίνει γιατί ο αλγόριθμος προβολής διατηρεί τις γραμμές αλλά όχι τις αποστάσεις. Από το σχήμα είναι εύκολο να το αντιληφθούμε αν υπολογίσουμε τον λόγο αποστάσεων μεταξύ V0P και $V0V1 = 4/6 = 0.666$ και V0'P' και $V0'V1' = 2/2.4 = 0.833$.

Έτσι πρέπει να «βρούμε» καινούριο αλγόριθμο που να λειτουργεί. Η λύση είναι να χρησιμοποιήσουμε τα αντίστροφα των συντεταγμένων στην εξίσωση των βαρυκεντρικών. Δηλαδή γίνεται $1/P.z = (1/V0.z) * \lambda 0 + (1/V1.z) * \lambda 1 + (1/V2.z) * \lambda 2$. Η συγκεκριμένη εξίσωση στο παραπάνω παράδειγμα δίνει:

$$1/P.z = (1/V0.z)(1 - \lambda) + (1/V1.z) * \lambda = ½ * (1 - 2/2.4) + 1/5 * (2/2.4) = 0.25.$$

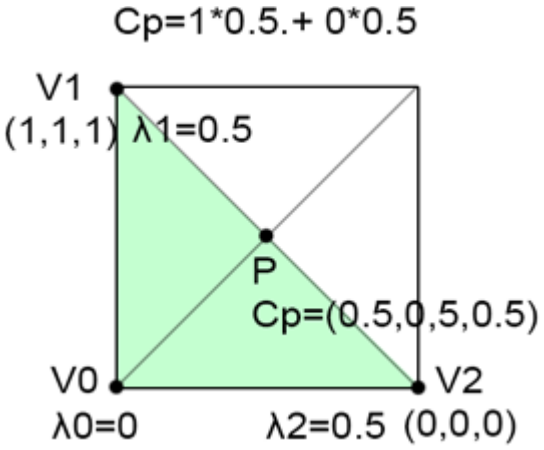
Το οποίο σημαίνει ότι $P.z = 1/0.25 = 4$. Αυτό είναι και το αναμενόμενο αποτέλεσμα. Η εξήγηση δεν είναι πολύ σύνθετη αλλά ξεπερνάει το σκοπό της διπλωματικής.

Προφανώς υπάρχουν και άλλοι αλγόριθμοι για την επίλυση του ζητήματος της ορατότητας. Χωρίζονται σε δυο κατηγορίες: 1) χώρου αντικειμένων (object space) και 2) χώρου εικόνας (image space). Ο Z-buffer ανήκει στην image space. Ο πιο «γνωστός» του object space είναι ο αλγόριθμος του ζωγράφου (painter's algorithm). Σε αυτόν τα στοιχεία ταξινομούνται με βάση την απόσταση τους και αρχικά «ζωγραφίζονται» τα πιο μακρινά, έπειτα τα πιο κοντινά κ.ο.κ. Αναφέραμε όμως το πρόβλημα που προκύπτει με αυτήν την λογική αναλύοντας τον Z-Buffer. Αξίζει να ειπωθεί ότι ο πιο κοινός αλγόριθμος είναι ο Z-Buffer ο οποίος χρησιμοποιείται από τις GPUs για αυτό και είναι αυτό που ανέλυσα.

2.1.7 Perspective Correct Interpolation (Αλγόριθμος «σωστής» προοπτικής)

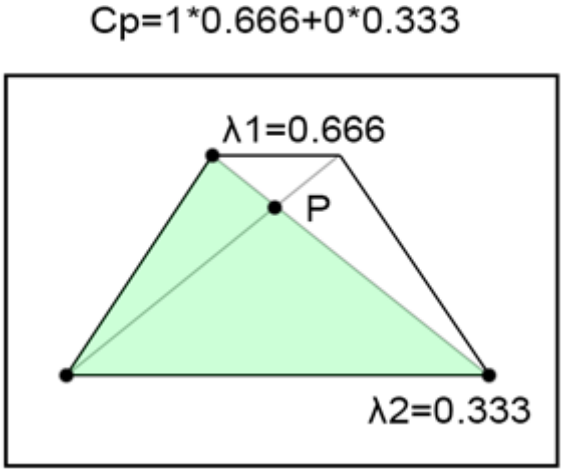
Για την λειτουργία ενός απλού rasterizer, οι αλγόριθμοι που αναφέραμε λύνουν τα βασικά προβλήματα, δηλαδή το πρόβλημα της κρυμμένης επιφάνειας και αυτό της επίλυσης αν το pixel είναι εντός ή εκτός τριγώνου. Σε περίπτωση όμως που επιθυμούμε ένα πιο αληθοφανές αποτέλεσμα ιδιαίτερα στη σκίαση (shading) και στον φωτισμό (lighting), τότε πρέπει αναφέρουμε τον perspective correct interpolation algorithm. Όπως αναφέραμε προηγουμένως, ο barycentric interpolation algorithm υπολογίζει τις διάφορες ιδιότητες βασιζόμενος στον τύπο $CP = \lambda 0 * C0 + \lambda 1 * C1 + \lambda 2 * C2$ και μπορεί να εφορμοστεί είτε στο χρώμα, είτε στα textures είτε στις νόρμες (normals) είτε (με μια παραλλαγή) στον Z-Buffer. Όμως ιδιαίτερα στα textures η τεχνική αυτή δυστυχώς δεν λειτουργεί. Για να κατανοήσουμε καλύτερα το πρόβλημα ακολουθεί ένα παράδειγμα.

Στην παρακάτω εικόνα, παρατηρούμε ένα σημείο στη μέση ενός 3D τετραγώνου. Η οπτική γωνία όπως μπορούμε να διακρίνουμε είναι κάθετα από πάνω και μπορούμε να καταλάβουμε ότι το σημείο P είναι ακριβώς στο κέντρο της πλευράς V1V2. Αν παρατηρήσουμε τις τιμές σε αυτό το σημείο των βαρυκεντρικών, κατανοούμε ότι λ0=0 και λ1=λ2=0.5.



Εικόνα 21: Κάθετη οπτική τετραγώνου

Αλλάζοντας όμως οπτική γωνία για το ίδιο ακριβώς τετράγωνο, παρατηρούμε ότι φαινομενικά το σημείο P δεν βρίσκεται πλέον στο μέσον της ακμής V1V2. Βρίσκεται περίπου στο 1/3 της απόστασης V1V2 οπότε και το λ1=0.66 και το λ2=0.333. Ο

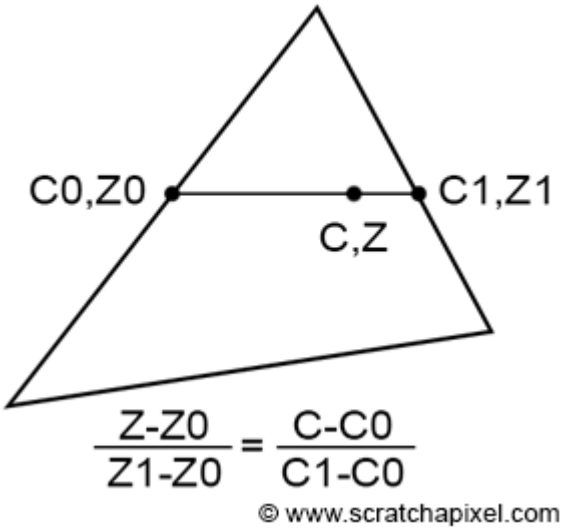


Εικόνα 22: Πρόβλημα προοπτικής τετραγώνου

αλγόριθμος των βαρυκεντρικών δεν εφαρμόζεται στον 3D χώρο αλλά στην προβολή του στην επιφάνεια της οθόνης. Άρα καθόλη τη διάρκεια της εφαρμογής του αλγορίθμου, θα εφαρμόζονται εσφαλμένες τιμές βαρών σε σχέση με τις 3D τιμές τους. Όπως εξηγήσαμε και στον Z-Buffer το πρόβλημα αυτό έγκειται στο γεγονός ότι ο αλγόριθμος προοπτικής δεν «διατηρεί» τις αποστάσεις.

Για την «επίλυση» του προβλήματος θα χρησιμοποιήσουμε τα αποτελέσματα του Z-Buffer ώστε να «κανονικοποιήσουμε» το αποτέλεσμα. Πιο αναλυτικά:

Έστω ότι θέλουμε να υπολογίσουμε το C της εικόνας που θα μπορούσε να είναι ένα οποιοδήποτε vertex attribute. Εφόσον και το Z και το C θα το υπολογίσουμε με linear interpolation, μπορούμε να γράψουμε το εξής:



Εικόνα 23: Υπολογισμός Z

$\frac{Z-Z_0}{Z_1-Z_0} = \frac{C-C_0}{C_1-C_0}$. Στον αλγόριθμο του Z-Buffer αν και δεν ανέφερα αναλυτικά τις πράξεις, είχαμε καταλήξει στην εξίσωση:

$$Z = \frac{1}{\frac{1}{Z_0}(1-q)+\frac{1}{Z_1}q}$$

Αν αντικαταστήσουμε λοιπόν τις εξισώσεις και λύσουμε ως προς C, καταλήγουμε στο $C = \frac{C_0 Z_1(1-q)+C_1 Z_0 q}{Z_1(1-q)+Z_0 q}$ όπου αν πολλαπλασιάσουμε κατά μέλη με $1/Z0Z1$, καταλήγουμε στο:

$C = Z[\frac{C_0}{Z_0}(1-q)+\frac{C_1}{Z_1}q]$. Ανέφερα μόνο τις εξισώσεις που χρειαζόμαστε και τα αποτελέσματά τους, καθώς τα ενδιάμεσα στάδια ξεπερνούν τον σκοπό της εργασίας. Αυτό που μας «εξηγούν» τα μαθηματικά του αλγορίθμου, είναι ότι προκειμένου να υπολογίσουμε τη σωστή τιμή μια ιδιότητας, πρέπει να διαιρέσουμε την τιμή αυτή με το βάθος που βρίσκεται το σημείο του διανύσματος, και μετά να εκτελέσουμε γραμμικό interpolation χρησιμοποιώντας τις βαρυκεντρικές συντεταγμένες (εδώ q προηγουμένως λ) και τέλος να πολλαπλασιάσουμε με το Z, που αντιστοιχεί στο βάθος του σημείου στο τρίγωνο.

Μία περιγραφή του αλγορίθμου αυτού (και παράλληλα του γραμμικού interpolation) βλέπουμε στο παρακάτω C++ πρόγραμμα.

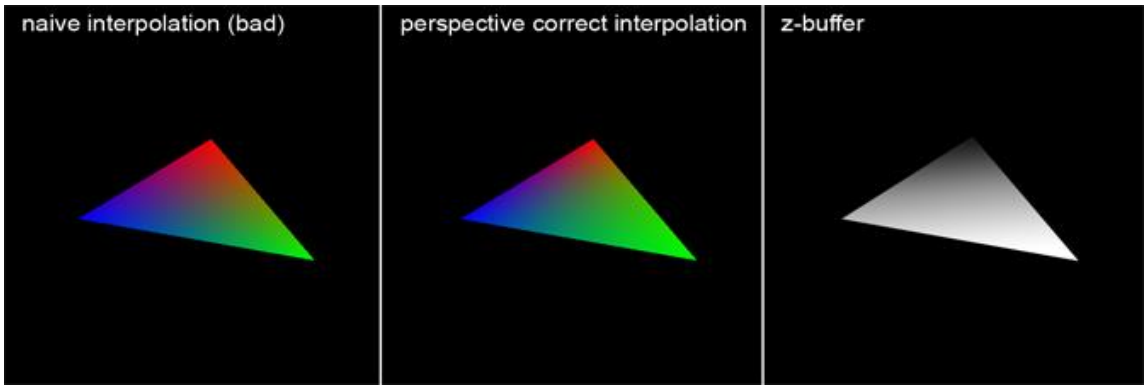
```
#include <cstdio>

#include <cstdlib>

#include <fstream>


typedef float Vec2[2];

typedef float Vec3[3];

typedef unsigned char Rgb[3];
```

```
inline
float edgeFunction(const Vec3 &a, const Vec3 &b, const Vec3 &c)
{ return (c[0] - a[0]) * (b[1] - a[1]) - (c[1] - a[1]) * (b[0] - a[0]); }

int main(int argc, char **argv)
{
    Vec3 v2 = { -48, -10,  82};
    Vec3 v1 = {  29, -15,  44};
    Vec3 v0 = {  13,  34, 114};
    Vec3 c2 = {1, 0, 0};
    Vec3 c1 = {0, 1, 0};
    Vec3 c0 = {0, 0, 1};

    const uint32_t w = 512;
    const uint32_t h = 512;

// project triangle onto the screen
    v0[0] /= v0[2], v0[1] /= v0[2];
    v1[0] /= v1[2], v1[1] /= v1[2];
    v2[0] /= v2[2], v2[1] /= v2[2];
// convert from screen space to NDC then raster (in one go)
    v0[0] = (1 + v0[0]) * 0.5 * w, v0[1] = (1 + v0[1]) * 0.5 * h;
    v1[0] = (1 + v1[0]) * 0.5 * w, v1[1] = (1 + v1[1]) * 0.5 * h;
    v2[0] = (1 + v2[0]) * 0.5 * w, v2[1] = (1 + v2[1]) * 0.5 * h;

#ifdef PERSP_CORRECT
// divide vertex-attribute by the vertex z-coordinate
    c0[0] /= v0[2], c0[1] /= v0[2], c0[2] /= v0[2];
    c1[0] /= v1[2], c1[1] /= v1[2], c1[2] /= v1[2];
    c2[0] /= v2[2], c2[1] /= v2[2], c2[2] /= v2[2];
// pre-compute 1 over z
    v0[2] = 1 / v0[2], v1[2] = 1 / v1[2], v2[2] = 1 / v2[2];
#endif

    Rgb *framebuffer = new Rgb[w * h];
    memset(framebuffer, 0x0, w * h * 3);
```

```
    float area = edgeFunction(v0, v1, v2);

    for (uint32_t j = 0; j < h; ++j) {
        for (uint32_t i = 0; i < w; ++i) {
            Vec3 p = {i + 0.5, h - j + 0.5, 0};
            float w0 = edgeFunction(v1, v2, p);
            float w1 = edgeFunction(v2, v0, p);
            float w2 = edgeFunction(v0, v1, p);
            if (w0 >= 0 && w1 >= 0 && w2 >= 0) {
                w0 /= area;
                w1 /= area;
                w2 /= area;
                float r = w0 * c0[0] + w1 * c1[0] + w2 * c2[0];
                float g = w0 * c0[1] + w1 * c1[1] + w2 * c2[1];
                float b = w0 * c0[2] + w1 * c1[2] + w2 * c2[2];
#ifdef PERSP_CORRECT
                float z = 1 / (w0 * v0[2] + w1 * v1[2] + w2 * v2[2]);
// if we use perspective correct interpolation we need to
// multiply the result of this interpolation by z, the depth
// of the point on the 3D triangle that the pixel overlaps.
                r *= z, g *= z, b *= z;
#endif
                framebuffer[j * w + i][0] = (unsigned char)(r * 255);
                framebuffer[j * w + i][1] = (unsigned char)(g * 255);
                framebuffer[j * w + i][2] = (unsigned char)(b * 255);
            }
        }
    }

    std::ofstream ofs;
    ofs.open("./raster2d.ppm");
    ofs << "P6\n" << w << " " << h << "\n255\n";
    ofs.write((char*)framebuffer, w * h * 3);
    ofs.close();

    delete [] framebuffer;
```

```
    return 0;
}
```

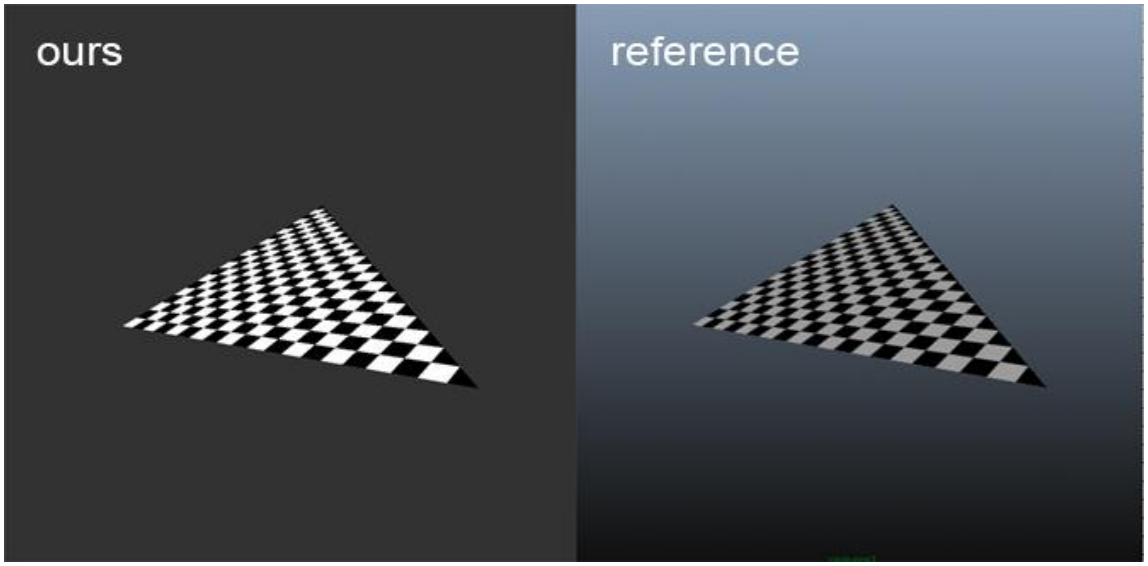
Έτσι λοιπόν «τρέχοντας» τον κώδικά μας ώστε να μπορέσουμε να συγκρίνουμε τα αποτελέσματα του αλγορίθμου παίρνουμε τις ακόλουθες εικόνες:



Εικόνα 24: Σύγκριση barycentric - perspective correct interpolation

Όπως εύκολα μπορούμε να αντιληφθούμε, η αριστερή εικόνα είναι η εικόνα του γραμμικού interpolation, ενώ η μεσσαία του perspective correct η οποία είναι πολύ πιο αληθοφανής. Δεξιά έχουμε την έξοδο του Z-Buffer όπου όσο πιο κοντά στην κάμερα είναι το σημείο, τόσο πιο φωτεινό είναι. Τα σύγχρονα API των γραφικών όπως το OpenGL και το Direct3D, όπως είναι φυσικό χρησιμοποιούν τον perspective correct. Η διαφορά είναι σαφώς πιο εμφανής αν προσπαθήσουμε να εφαρμόσουμε τους αλγόριθμους στο texturing (υφές).

Ανταλλάζοντας τα χρώματα ως attribute με τις texture συντεταγμένες παράδειγμα του της σκακίερας, θα πάρουμε το ακόλουθο αποτέλεσμα



Εικόνα 25: Επιβεβαίωση ορθής λειτουργίας του αλγορίθμου μέσω οπτικής σύγκρισης

Όπως παρατηρούμε, ο αλγόριθμος που περιγράψαμε δίνει αρκετά καλά αποτελέσματα οπτικά και δουλεύει. Ένα παράδειγμα ακόμα πιο οφθαλμοφανές, για την διαφορά μεταξύ 2 αλγορίθμων ειδικά στο texturing είναι αυτό που ακολουθεί:

Εικόνα 26: Διαφορά barycentric - perspective correct στο chess matrix

Όπως παρατηρούμε, εδώ η διαφορά είναι πολύ πιο εμφανής. Οπότε είναι αντιληπτό, γιατί ακόμα και GPUs χαμηλής κατανάλωση – απαιτήσεων (όπως αυτή που υλοποιήθηκε στην εργασία) χρησιμοποιούν τον perspective correct αλγόριθμο για τα textures ενώ για τα χρώματα τον γραμμικό.

2.1.8 Bresenham Algorithm

Ένας από τους πιο γνωστούς και διαδεδομένους αλγόριθμους για τη σχεδίαση **γραμμών** είναι ο Bresenham [6]. Το γεγονός που τον καθιστά πολύτιμο, είναι ότι εκτελείται αποκλειστικά σε ακεραίους (integers). Η λειτουργία του έχει ως εξής:

Έστω δυο σημεία της ευθείας:

P(x0,y0), P(x1,y1)

Έστω η εξίσωση της ευθείας υπολογίζεται ως:

$(x1-x0)(y-y0)-(x-x0)(y1-y0) = 0$ όπου ορίζουμε ως $dx = x1-x0$ και $dy = y1-y0$.

Έστω η συνάρτηση λάθους (απόκλεισης) που ορίζεται ως εξής:

$$e = (y-y0)dx-(x-x0)dy$$

Κατά μήκος της διαγωνίου το λάθος e_{xy} υπολογίζεται ως

$$exy = (y+1-y0)dx-(x+1-x0)dy = e+dx-dy$$

Κατά τον άξονα x ως e_x:

$$ex = (y+1-y0)dx-(x-x0)dy = exy+dy$$

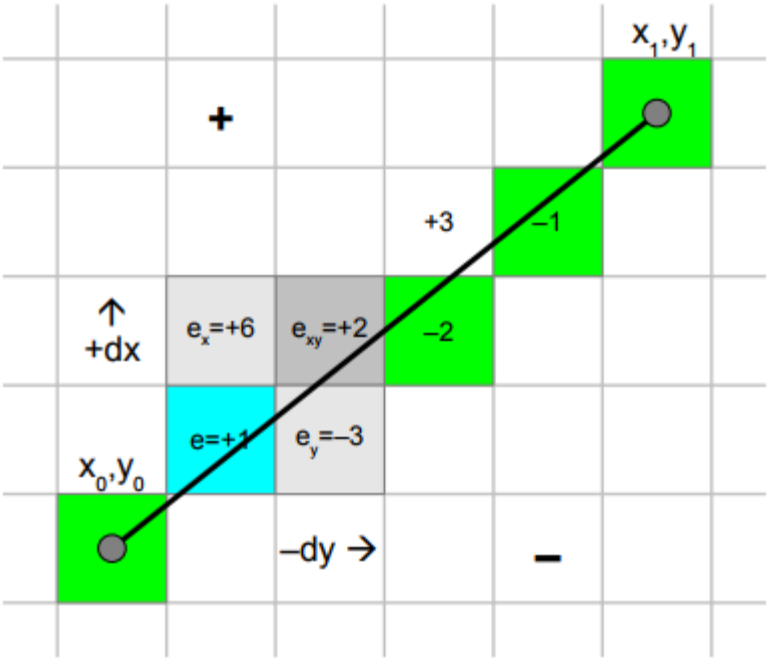
ενώ κατά τον άξονα y ως e_y:

$$ey = (y-y0)dx-(x+1-x0)dy = exy\ dx-dy$$

Το αρχικό λάθος e_1 ορίζεται ωε εξής:

$$e1 = (y0+1-y0)dx-(x0+1-x0)dy = dx-dy$$

Για καλύτερη κατανόηση παρουσιάζεται το ακόλουθο σχήμα εκτέλεσης του αλγορίθμου για dx=5 και dy=4.



Εικόνα 27: Λειτουργία Bresenham και υπολογισμός λαθών

Όπως παρατηρούμε, το γαλάζιο σχήμα έχει τη μικρότερη τιμή λάθους. Οι επόμενες πιθανές επιλογές χρωματίζονται με γκρι. Το πιο σκούρο έχει τη μικρότερη απόλυτη τιμή και υπολογίζεται 1 βήμα νωρίτερα καθώς το διαγώνιο pixel είναι πάντα ένα βήμα μπροστά στον αλγόριθμο. Εφόσον $exy + ey = +2 - 3 = -1 < 0$ η τιμή του y μειώνεται και προστίθεται dx στην τιμή του λάθους. Παρομοίως υπολογίζουμε και για το x $exy + ex = +2 + 6 = +8 > 0$ οπότε η τιμή του x αυξάνεται και αφαιρείται dy από την τιμή του λάθους.

Παρατήρηση: Παρόλο που φαίνεται ότι στο σχήμα οι άξονες x και y είναι ανάποδα κάτι τέτοιο δεν ισχύει. Απλά για τον υπολογισμό του λάθους χρησιμοποιούνται οι αντίστροφες μεταβολές. Ένας αλγόριθμος που υπολογίζει την εξίσωση της γραμμής με τον τρόπο που περιγράψαμε (Bresenham) είναι ο ακόλουθος.

```
void plotLine(int x0, int y0, int x1, int y1)
{
   int dx = abs(x1-x0), sx = x0<x1 ? 1 : -1;
   int dy = -abs(y1-y0), sy = y0<y1 ? 1 : -1;
   int err = dx+dy, e2; /* error value e_xy */
   for (;;){ /* loop */
      setPixel(x0,y0);
      e2 = 2*err;
      if (e2 >= dy) { /* e_xy+e_x > 0 */
         if (x0 == x1) break;
         err += dy; x0 += sx;
      }
      if (e2 <= dx) { /* e_xy+e_y < 0 */
         if (y0 == y1) break;
         err += dx; y0 += sy;
      }
   }
}
```

Αξίζει να σημειωθεί ότι ο αλγόριθμος Bresenham είναι έτσι σχεδιασμένος ώστε το λάθος στο τελευταίο pixel να είναι πάντα 0.

2.2 Ray Casting.

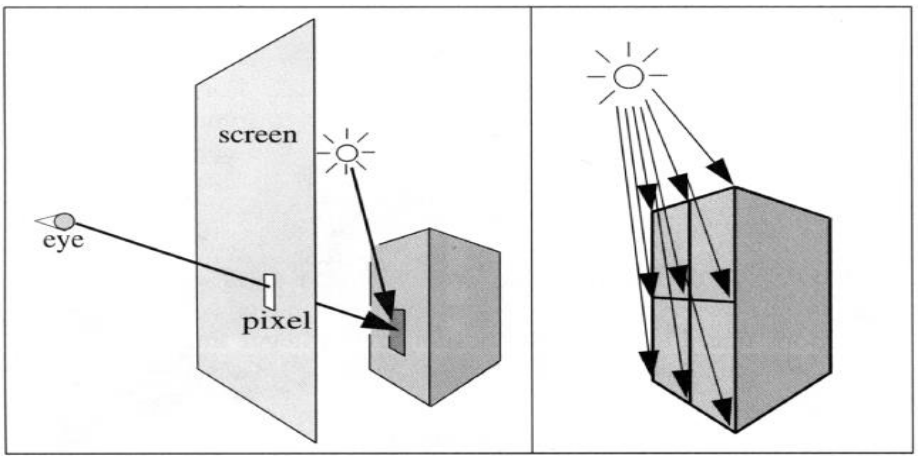
Ray casting ονομάζεται μία μέθοδος η οποία έχει σκοπό να λύσει μια σειρά προβλημάτων στον τομέα των γραφικών των ηλεκτρονικών υπολογιστών. Ο όρος χρησιμοποιήθηκε για πρώτη φορά σε ένα έγγραφο του 1982 από τον Scott Roth για να περιγράψει μια μέθοδο για την απόδοση τρισδιάστατων στερεών μοντέλων (CSG constructive solid geometry models).

Η μέθοδος ray casting δεν πρέπει να συγχέεται με την μέθοδο ray tracing, καθώς είναι δύο διαφορετικές μέθοδοι αλλά μπορεί να θεωρηθεί ως μια συνοπτική και σημαντικά ταχύτερη, έκδοση του ray tracing αλγόριθμου. Και οι δύο είναι αλγόριθμοι που χρησιμοποιούνται σε γραφικά υπολογιστών για να παρουσιάσουν τις τρισδιάστατες σκηνές σε δύο διαστάσεων στις οθόνες των υπολογιστών, ακολουθώντας τις ακτίνες του φωτός από το μάτι του παρατηρητή προς μια πηγή φωτός.

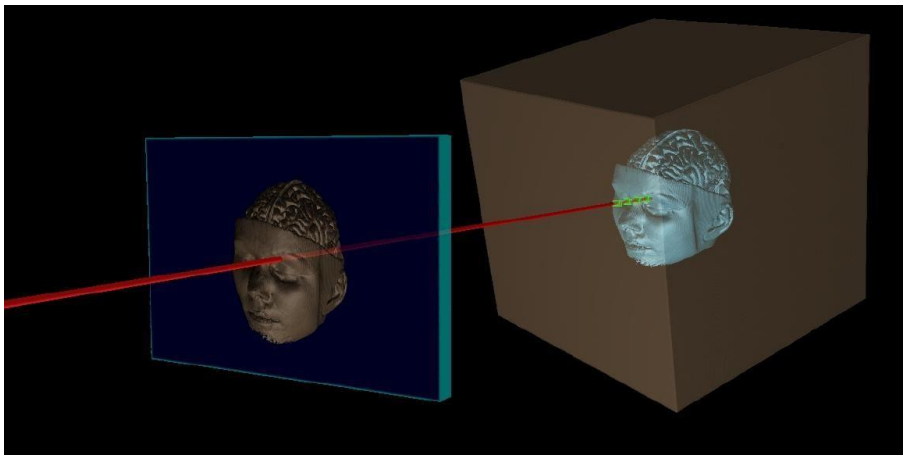
Στη φύση, μια πηγή φωτός εκπέμπει φωτεινές ακτίνες που ταξιδεύουν στο περιβάλλον. Στην διαδρομή των ακτινών παρεμβάλλονται επιφάνειες που διακόπτουν την πορεία τους. Σε αυτό το σημείο κάθε φωτεινή ακτίνα έχει τις εξής επιλογές, την απορρόφηση, την αντανάκλαση και τη διάθλαση στην εκάστοτε επιφάνεια. Η επιφάνεια μπορεί να αντανακλά το σύνολο ή μέρος της ακτίνας σε μία ή περισσότερες κατευθύνσεις. Θα μπορούσε να απορροφήσει και μέρος του φωτός με αποτέλεσμα την απώλεια της έντασης της ανακλώμενης ή διαθλώμενης ακτίνας. Αν η επιφάνεια έχει οποιαδήποτε διαφανή ή ημιδιαφανή ιδιότητα τότε διαθλά ένα μέρος της φωτεινής δέσμης. Μεταξύ απορρόφησης, ανάκλασης, και διάθλασης, όλες οι εισερχόμενες ακτίνες φωτός πρέπει να λαμβάνονται υπόψη στον υπολογισμό του αλγορίθμου. Ορισμένες από αυτές τις ακτίνες ταξιδεύουν με τέτοιο τρόπο ώστε τελικά προσπίπτουν στα μάτια του παρατηρητή, κάνοντάς τον έτσι να δει τη σκηνή. Ο αλγόριθμος ray casting προσπαθεί να μιμηθεί τη διαδικασία αυτή με τη χρήση ηλεκτρονικού υπολογιστή. Το ελάττωμα της μεθόδου είναι ότι μπορεί να θεωρηθεί εξαιρετικά σπάταλη, δεδομένου ότι μόνο ένα μικρό μέρος της φωτεινής ακτίνας θα φτάσει πράγματι στο μάτι του παρατηρητή.

Η βασική ιδέα της μεθόδου ray casting είναι ότι ξεκινούν οι φωτεινές ακτίνες από το μάτι του παρατηρητή, μία για κάθε pixel, που θα βρίσκει το κοντινότερο αντικείμενο που μπλοκάρει το δρόμο της. Στη συνέχεια το μάτι βλέπει κανονικά μέσω του pixel.

Χρησιμοποιώντας τις ιδιότητες των υλικών και την επίδραση των φωτεινών πηγών στην σκηνή, ο αλγόριθμος αυτός μπορεί να προσδιορίσει τη σκίαση αυτού του αντικειμένου. Η σκίαση της επιφάνειας υπολογίζεται χρησιμοποιώντας την παραδοσιακή μέθοδο σκίασης των τρισδιάστατων μοντέλων. Ένα σημαντικό πλεονέκτημα της ray casting μεθόδου είναι ότι αντιμετωπίσει εύκολα μη επίπεδες επιφάνειες και στερεά. Ο ray casting αλγόριθμος **δεν** υπολογίζει τη νέα κατεύθυνση των ακτινών συμπεριλαμβάνοντας τις διαθλάσεις και αντανακλάσεις της ακτίνας στις επιφάνειες των αντικειμένων. Η υψηλή ταχύτητα των υπολογισμών της μεθόδου ray casting βοηθούν στην εφαρμογή φωτορεαλιστικών αποδόσεων στα real-time 3D video games. Στις επόμενες εικόνες παρατηρούμε τον τρόπο λειτουργίας του αλγορίθμου.



Εικόνα 28: Απεικόνιση του τρόπου λειτουργίας της μεθόδου Ray Casting



Εικόνα 29: Απεικόνιση ενός τρισδιάστατου αντικειμένου στην δισδιάστατη οθόνη του υπολογιστή

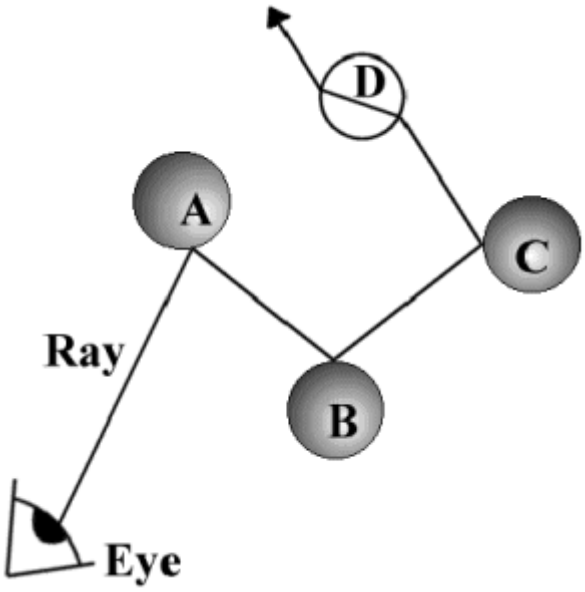
2.3 Ray Tracing.

Η ray tracing είναι μία μέθοδος που έχει ως στόχο να κατασκευάζει εικόνες προερχόμενες από εικονικά τρισδιάστατα μοντέλα μίας σκηνής. Λειτουργεί με την ανίχνευση της διαδρομής μιας φωτεινής ακτίνας, που παρατηρείται μέσα από ένα φανταστικό μάτι σε κάθε pixel μιας οθόνης υπολογιστή, και υπολογίζει το χρώμα και την σκίαση του αντικειμένου. Οι σκηνές περιγράφονται μαθηματικά και ενσωματώνουν όλα τα τρισδιάστατα μοντέλα που έχουν κατασκευαστεί. Στον ray tracing αλγόριθμο εκτοξεύονται ακτίνες φωτός όπως και στην μέθοδο ray casting, η διαφορά έγκειται στο ότι συνεχίζει να υπολογίζει την διαδρομή του φωτός και μετά την πρόσκρουση στο πρώτο αντικείμενο (επιφάνεια). Μόλις το πλησιέστερο αντικείμενο έχει εντοπιστεί, ο αλγόριθμος θα εκτιμήσει το εισερχόμενο φως στο σημείο πρόσκρουσης, θα εξετάσει τις ιδιότητες του υλικού του αντικειμένου και θα συνδυάζει τα στοιχεία για να υπολογίσει το τελικό χρώμα του pixel.

Όπως αναφέρθηκε και στην μέθοδο ray casting μπορεί να φαίνεται αντιφατικό οι ακτίνες να εκτοξεύονται από τον παρατηρητή προς τα αντικείμενα και έτσι να αναγνωρίζεται η θέση της φωτεινής πηγής (το αντίθετο δηλαδή με αυτό που το φως κάνει στην πραγματικότητα), αλλά κάτι τέτοιο είναι πολύ πιο αποτελεσματικό, δεδομένου ότι η συντριπτική πλειοψηφία των ακτινών φωτός από μια δεδομένη φωτεινή πηγή δεν καταγράφονται γιατί δεν φτάνουν ποτέ στο μάτι του παρατηρητή.

Ένας έξυπνος περιορισμός της μεθόδου είναι ότι μετά από ένα ορισμένο μέγιστο αριθμό ανακλάσεων ή ορισμένης απόστασης στην οποία η ακτίνα δεν έχει συναντήσει καμία επιφάνεια, τότε παύει να ταξιδεύει και η τιμή του pixel θεωρείται ενημερωμένη. Η ένταση του φωτός αυτού του pixel υπολογίζεται χρησιμοποιώντας μια σειρά άλλων αλγορίθμων, που μπορεί να περιλαμβάνουν και άλλες τεχνικές όπως την μέθοδο radiosity. Ο αριθμός των ανακλάσεων αλλάζει κάθε φορά που η ακτίνα συναντά μια επιφάνεια. Συνήθως κάθε ακτίνα επιτρέπεται να ανακλαστεί μέχρι και 16 φορές ενώ ο αριθμός των διαθλάσεων μια ακτίνας ομοίως είναι 9 φορές.

Η μέθοδος αυτή μπορεί να παράγει πολύ μεγάλου βαθμού οπτικό ρεαλισμό, καλύτερο των τυπικών μεθόδων απόδοσης (scanline κ.α.) αλλά με μεγαλύτερο υπολογιστικό κόστος. Το γεγονός αυτό καθιστά την μέθοδο ray tracing καταλληλότερη για εφαρμογές όπου η εικόνα μπορεί να κατασκευαστεί σιγά-σιγά, όπως στην φωτογραφία, στον κινηματογράφο και τα ειδικά εφέ, άλλα δεν ενδείκνυται για εφαρμογές πραγματικού χρόνου, όπως τα βιντεοπαιχνίδια, όπου η ταχύτητα είναι κρίσιμη συνιστώσα. Η Ray tracing μέθοδος είναι ικανή να προσομοιώσει μια ευρεία ποικιλία οπτικών φαινομένων, όπως η αντανάκλαση, η διάθλαση, η σκέδαση και η χρωματική εκτροπή. Εκτός από τον υψηλό βαθμό ρεαλισμού μπορεί να προσομοιώσει και τα οπτικά αποτελέσματα μιας κάμερας. Χρησιμοποιώντας τις ιδιότητες των υλικών και την επίδραση των φωτεινών ακτινών στην σκηνή, ο αλγόριθμος αυτός μπορεί να καθορίσει και τη σκίαση του αντικειμένου χρησιμοποιώντας και άλλες παραδοσιακές τεχνικές σκίασης. Ένα σημαντικό πλεονέκτημα που προσφέρει σε σχέση με τους αλγόριθμους scanline είναι η ικανότητά της μεθόδου να αντιμετωπίσει εύκολα μη επίπεδες επιφάνειες και στερεά, όπως κώνους και σφαίρες. Περίτεχνα αντικείμενα μπορούν να αποδοθούν εύκολα με τη χρήση της μεθόδου ray tracing. Στη συνέχεια βλέπουμε τον τρόπο με τον οποίο λειτουργεί ο αλγόριθμος:



Εικόνα 30: Απεικόνιση του τρόπου λειτουργίας της μεθόδου Ray Tracing

Τα πλεονεκτήματα έναντι άλλων μεθόδων απόδοσης.

Η μέθοδος ray tracing είναι η δημοφιλέστερη σε μη real-time εφαρμογές, λόγω της ρεαλιστικής προσομοίωσης του φωτισμού σε σχέση με άλλες μεθόδους απόδοσης (όπως η scanline ή η ray casting). Τα αποτελέσματα που φέρει όπως αντανακλάσεις και σκιές , τα οποία είναι δύσκολο να προσομοιωθούν με άλλες μεθόδους , είναι φυσικά αποτελέσματα του ray tracing αλγόριθμου. Πλέον θεωρείται η καλύτερη μέθοδος για φωτορεαλιστικές αποδόσεις και τα κορυφαία προγράμματα χρησιμοποιούν renderers και αλγόριθμους που αξιοποιούν στο έπακρο την μέθοδο αυτή.

Η επόμενη εικόνα έχει κατασκεπαστεί αποκλειστικά με τη μέθοδο ray tracing και δεν χρησιμοποιεί καμία φωτογραφία κανενός αντικειμένου [7]:

Εικόνα 31: Φωτορεαλιστική εικόνα μέσω Ray Casting

Τα μειονεκτήματα έναντι άλλων μεθόδων απόδοσης.

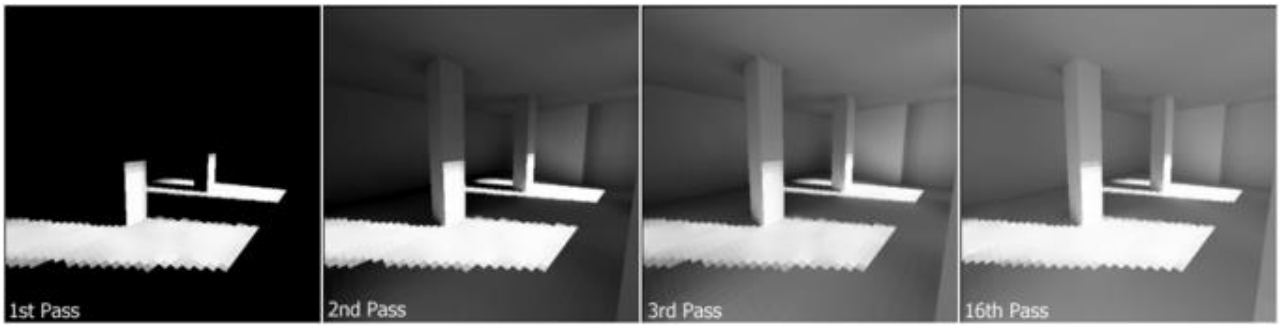
Ένα σοβαρό μειονέκτημα του ray tracing είναι ο χρόνος απόδοσης. Άλλοι αλγόριθμοι χρησιμοποιούν τα δεδομένα της συνοχής με στόχο να μοιράσουν τους υπολογισμούς μεταξύ των pixels, ενώ αντίθετα η ray tracing μέθοδος ξεκινά τη διαδικασία εκ νέου. Ωστόσο αυτή η διαφορά προσφέρει άλλα πλεονεκτήματα, όπως η βελτίωση της ποιότητας της εικόνας. Η εφαρμογή της εξίσωσης της απόδοσης (εξίσωση rendering) δίνει πραγματικό φωτορεαλισμό καθώς περιγράφει κάθε φυσική επίδραση του φωτός πάνω στις επιφάνειες. Ωστόσο, αυτό είναι συνήθως ανέφικτο λόγω των υπολογιστικών πόρων που απαιτούνται.

2.4 Radiosity.

Αποτελεί μια διαφορετική μέθοδο σε σχέση με τις προαναφερθείσες καθώς είναι αλγόριθμος γενικού φωτισμού (global illumination algorithm) [8]. Για να γίνει πιο κατανοητό, δεν ακολουθεί μόνο την πορεία μιας ακτίνας από την πηγή του φωτός, αλλά ακολουθεί και την φωτεινότητα από άλλες επιφάνειες που αντανακλούν το φως δρώντας ως πηγές φωτός στο προς εξέταση αντικείμενο. Δηλαδή λαμβάνει υπόψιν την αντανάκλαση και τη διάχυση του φωτός πάνω στις επιφάνειες και τον τρόπο που με τη σειρά τους επηρεάζουν τις υπόλοιπες επιφάνειες. Αυτή η διαδικασία παράγει πολύ ρεαλιστικές σκιές και αντικατοπτρισμούς ιδίως σε μία σκηνή εσωτερικού χώρου. Κλασικό παράδειγμα ο τρόπος σκίασης των γωνιών ενός δωματίου. Ένα χαρακτηριστικό της μεθόδου είναι ότι δεν εξαρτάται από την θέση του παρατηρητή (viewpoint). Αυτό σημαίνει ότι γίνονται πολύ περισσότεροι υπολογισμοί από ότι είναι απαραίτητοι, αλλά τα αποτελέσματά της μπορούν να χρησιμοποιηθούν για διαφορετικά σημεία θέασης.

Για την εφαρμογή του αλγορίθμου, η προς επεξεργασία εικόνα διαιρείται σε μικρότερες επιφάνειες (patches). Για κάθε ζεύγος αυτών των επιφανειών, υπολογίζεται ένας παράγοντας ανάκλασης (view-form factor) δηλαδή το ποσοστό του φωτός που ανακλάται από την επιφάνεια Α και «φωτίζει» την επιφάνεια Β. Απομακρυσμένες επιφάνειες και επιφάνειες με «λοξές» γωνίες, θα έχουν μικρότερο form factor από κοντινές. Οι παράγοντες αυτοί χρησιμοποιούνται ως παράγοντες (coefficients) σε ένα γραμμικό σύστημα επίλυσης rendering. Έτσι, στο προοδευτικό radiosity, μετά από κάθε επανάληψη υπολογισμού των ζευγών, θα έχουμε και ένα πιο φωτορεαλιστικό αποτέλεσμα. Ο χρήστης μπορεί να σταματήσει ανά πάσα στιγμή αν στο συγκεκριμένο πέρασμα, ο αλγόριθμος έχει παράξει αποδεκτό αποτέλεσμα.

Στην παρακάτω εικόνα παρατηρούμε τη βηματική απόδοση του αλγορίθμου



Εικόνα 32: Οπτικά αποτελέσματα για διαφορετικό αριθμό βημάτων Radiosity

Ο αλγόριθμος radiosity, αν και απλός στην κατανόησή του, έχει στην βασική του μορφή τετραγωνική πολυπλοκότητα. Αν όμως το εξετάσουμε ως ένα πρόβλημα υφής της σκηνής τότε μπορούμε να πάμε σε γραμμική πολυπλοκότητα. Επειδή το αποτέλεσμα του

αλγορίθμου είναι οπτικά αρκετά καλό, πολλοί επαγγελματίες στον τομέα το γραφικών χρησιμοποιούν μια τεχνική που αποκαλούν false radiosity. Η λογική της λειτουργίας της είναι να προσπαθήσουν να μιμηθούν τον αλγόριθμο, σκουραίνοντας γωνίες, εσοχές και σημεία επαφής σε εικόνες που χρησιμοποιούν για texturing και εφαρμόζοντάς τα με διάχυση πάνω στη σκηνή χρησιμοποιώντας έναν renderer όπως ο scanline (αλγόριθμος rasterizing).

Εικόνα 33: Απεικόνιση της λειτουργίας της μεθόδου Radiosity και βελτίωσης της σκίασης από α) εφαρμογή απευθείας φωτισμού σε β) με την εφαρμογή της μεθόδου radiosity

3. FPGA

3.1 Εισαγωγή στα FPGA

Το FPGA (Field Programmable Gate Array) είναι ένα ολοκληρωμένο κύκλωμα (Integrated Circuit), το οποίο επιτρέπει στο χρήστη σχεδιάσει ένα ολοκληρωμένο σύστημα, χρησιμοποιώντας την κατάλληλη γλώσσα προγραμματισμού. Σε αντίθεση με το ASIC το FPGA είναι επαναπρογραμματίσιμο γεγονός που το καθιστά ιδανικό για εφαρμογές που δεν έχουν οριστικοποιηθεί και στις οποίες οι απαιτήσεις και οι στόχοι συνεχώς αλλάζουν. Το κόστος του σε σύγκριση με την παραγωγή ASIC είναι ελάχιστο ενώ η απόδοση. Το FPGA αποτελείται από Blocks προγραμματιζόμενης λογικής (logic blocks) και προγραμματιζόμενες διασυνδέσεις (interconntects). Εκτός από αυτά τα στοιχεία, όλα τα FPGAs έχουν μνήμες RAM και διαφορετικό πλήθος και είδος από εισόδους-εξόδους (IOs) καθώς και διάφορους μετατροπείς αναλογικού-ψηφιακού (ADC, DAC) [9].

3.2 Ιστορική Αναδρομή

Ως «πρόγονοι» του FPGA μπορούν να θεωρηθούν οι απλές Συσκευές Προγραμματιζόμενης Λογικής (Programmable Logic Devices - PLDs) και η Προγραμματιζόμενη Μνήμη μόνο-για-ανάγνωση (Programmable Read-Only Memory PROM). Η PROM ήταν μια μη-πτητική (non-volatile) μνήμη, η οποία μπορούσε να φορτωθεί με δεδομένα. Μπορούσε να προγραμματιστεί είτε στο επίπεδο εργοστασιακής κατασκευής είτε στο επίπεδο του χρήστη (field-programmable). Τα PLDs ήταν ηλεκτρονικές συσκευές, οι οποίες περιείχαν ένα σύνολο από λογικές πύλες OR και AND και μπορούσαν επίσης να προγραμματιστούν όπως η PROM.

Το 1985 οι συνιδρυτές της Xilinx, R. Freeman και B. Vonderschmitt παρουσίασαν το πρώτο εμπορικό FPGA, το XC2064 [25]. Το XC2064 περιείχε προγραμματιζόμενες πύλες και προγραμματιζόμενες διασυνδέσεις μεταξύ πυλών και όρισε τις απαρχές μια νέας τεχνολογίας και αγοράς.

Τη δεκαετία του ’90, η παραγωγή FPGA αυξήθηκε ραγδαία. Εμφανίστηκαν και άλλες εταιρίες και το ποσοστό της αγοράς μοιράστηκε. Στις αρχές της δεκαετίας, τα FPGAs χρησιμοποιήθηκαν κυρίως σε εφαρμογές τηλεπικοινωνιών και δικτύων, ενώ αργότερα επεκτάθηκαν και στη βιομηχανία και αυτοκινητοβιομηχανία. Στο μέλλον προβλέπεται να χρησιμοποιηθούν επικουρικά με τους επεξεργαστές (Η Intel έχει εξαγοράσει την Altera) ώστε να επιτυγχάνουν αύξηση στην απόδοση συγκεκριμένων εφαρμογών ανάλογα τον χρήστη, διατηρώντας χαμηλή κατανάλωση ενέργειας αλλά και τη δυνατότητα επαναπρογραμματισμού σε περίπτωση που αλλάξουν οι απαιτήσεις, ή υπάρξει βελτιωμένη λύση για την συγκεκριμένη εργασία.

Ιστορικά, τα FPGAs υπήρξαν πιο αργά, κατανάλωναν περισσότερη ενέργεια και πετύχαιναν λιγότερη λειτουργικότητα σε σχέση με τα αντίστοιχα ASICs. Όμως, με την πάροδο των χρόνων, με την ανάπτυξη της τεχνολογίας επιτεύχθηκε:

- Αύξηση της αξιοπιστίας τους,
- Αύξηση των παρεχόμενων πυλών
- Αύξηση της ταχύτητας του ρολογιού καθώς
- Αύξηση εισόδων-εξόδων
- Μειωμένες ενεργειακές απαιτήσεις
- Μείωση κόστους
- Δυνατότητα σχεδιασμού πολυπλοκότερων chip
- Επανασχεδιασμό “on-the-fly” (ενώ λειτουργεί μέρος του FPGA)

Όλα τα προηγούμενα σε συνδυασμό με τη δυνατότητα διόρθωσης λαθών (bugs) αλλά και γρήγορης παραγωγής (time to market) οδηγούν αρκετούς κατασκευαστές τουλάχιστον

στα πρώτα στάδια υλοποίησης να τα χρησιμοποιούν όλο και περισσότερο τουλάχιστον στην αρχή (ASIC prototyping σε FPGA).

Σύμφωνα με τις εκτιμήσεις της Xilinx, η ανάπτυξη των FPGA στην αγορά τα τελευταία χρόνια έχει ευνοηθεί από παράγοντες όπως:

- Αύξηση του κόστους ανάπτυξης ολοκληρωμένων κυκλωμάτων
- Επιμήκυνση του απαιτούμενου χρόνου σχεδίασης λόγω της πολυπλοκότητας των ASIC
- Μείωση των πόρων έρευνας και ανάπτυξης
- Αυξημένες απώλειες λόγω αργού χρόνου παράδοσης στην αγορά
- Οικονομικοί περιορισμοί, οι οποίοι προωθούν τεχνολογίες χαμηλού κόστους

Αυτές οι τάσεις έχουν καταστήσει τα FPGA ως μια πολύ ευέλικτη εναλλακτική έναντι των ASICs σε πολύ περισσότερες εφαρμογές σε σχέση με το παρελθόν. Αυτή η εξέλιξη συνεπικουρείται από το γεγονός ότι ο αριθμός πυλών σε ένα FPGA έχει αυξηθεί ραγδαία από μερικές χιλιάδες τη δεκαετία του ΄80 σε εκατομμύρια πύλες μετά το 2000.

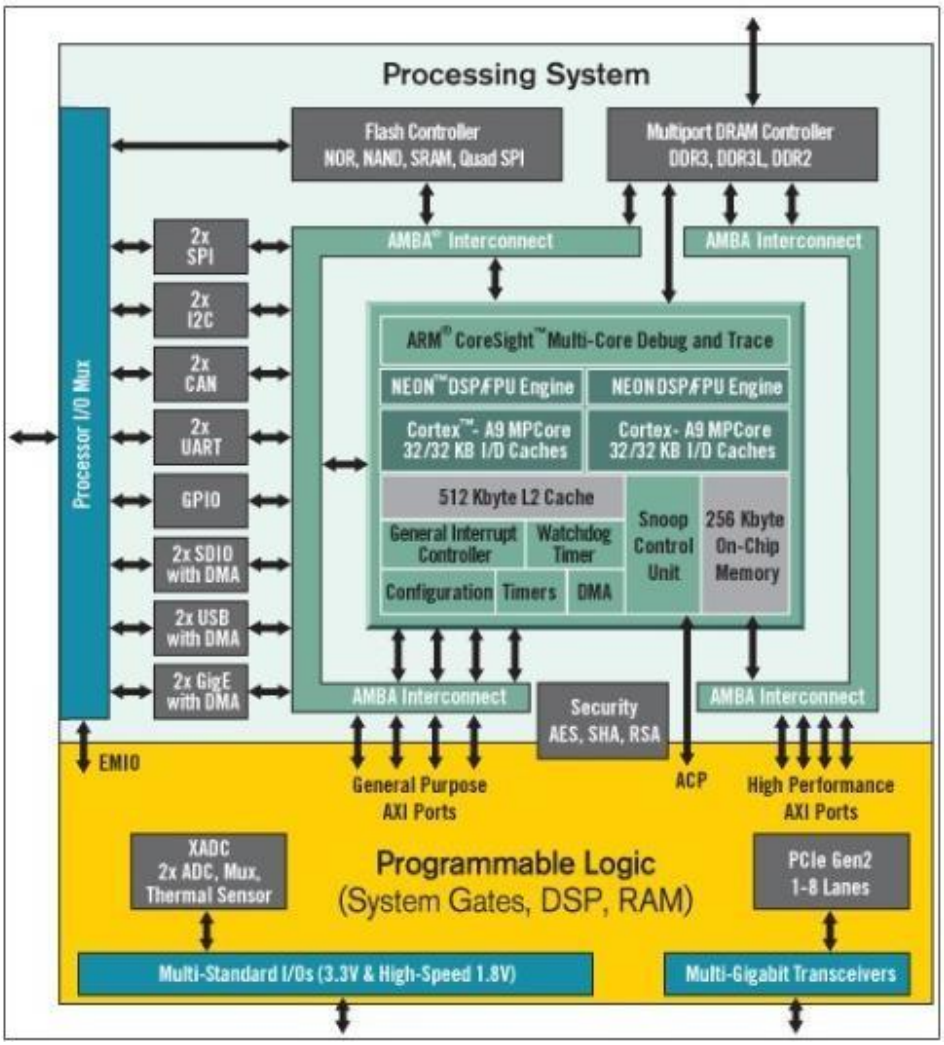
Ως αποτέλεσμα, το μέγεθος της αγοράς των FPGAs έχει μεγαλώσει πάρα πολύ και εκτιμάται ότι μέχρι το 2020 θα αγγίξει τα $10 δις. Τη μερίδα του λέοντος (τουλάχιστον 90%) μοιράζονται οι δύο μεγάλες εταιρίες, η Xilinx και η Altera.

Μια σύγχρονη τάση είναι ο συνδυασμός, των λογικών μπλοκ και των διασυνδέσεων ενός «παραδοσιακού» FPGA, με ενσωματωμένους μικροεπεξεργαστές και σχετικά περιφερειακά, για το σχηματισμό ενός πλήρους συστήματος σε προγραμματιζόμενο τσιπ (Programmable System on Chip). Η αρχή έγινε με τη σειρά Zynq της Xilinx ανοίγοντας μια ολόκληρη αγορά και διευρύνοντας το «κοινό» που απευθυνόντουσαν τα FPGAs.

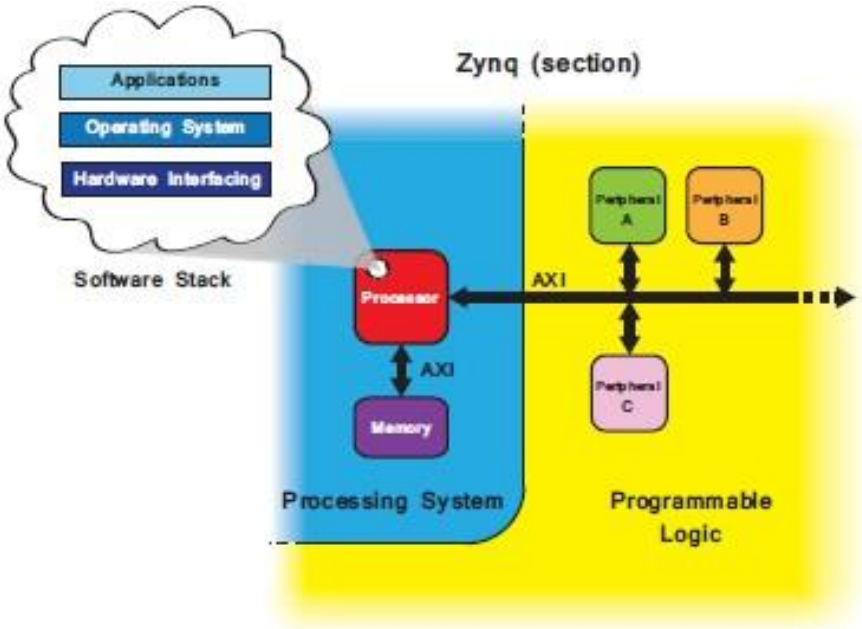
3.3 Το Zynq-7000 (ZC702)

Το ZC702 είναι ένα FPGA της σειράς Zynq-7000 το οποίο περιέχει 2 επεξεργαστές ARM Cortex A9 – MPCore [10]. Όπως αναφέρθηκε η ενσωμάτωση του Arm βοηθάει στο συνδυασμό προγραμματισμού σε γλώσσα υψηλού επιπέδου και την δημιουργία accelerators με τη χρήση γλώσσας RTL (VHDL, Verilog) ή ακόμα και υψηλού επιπέδου (C, C++, SystemC). Επιπλέον τα σφάλματα διασύνδεσης ελαχιστοποιούνται, τα οποία αποτελούν και τα συνήθη σφάλματα στις πλακέτες PCB όπου χρησιμοποιούνται σε αντίστοιχες εφαρμογές. Στη συνέχεια θα κάνουμε μια σχετικά σύντομη αλλά περιεκτική παρουσίαση του Zynq 7000.

Στη συνέχεια βλέπουμε τη δομή ενός FPGA της σειράς Zynq 7000



Εικόνα 34: Αρχιτεκτονική Zynq

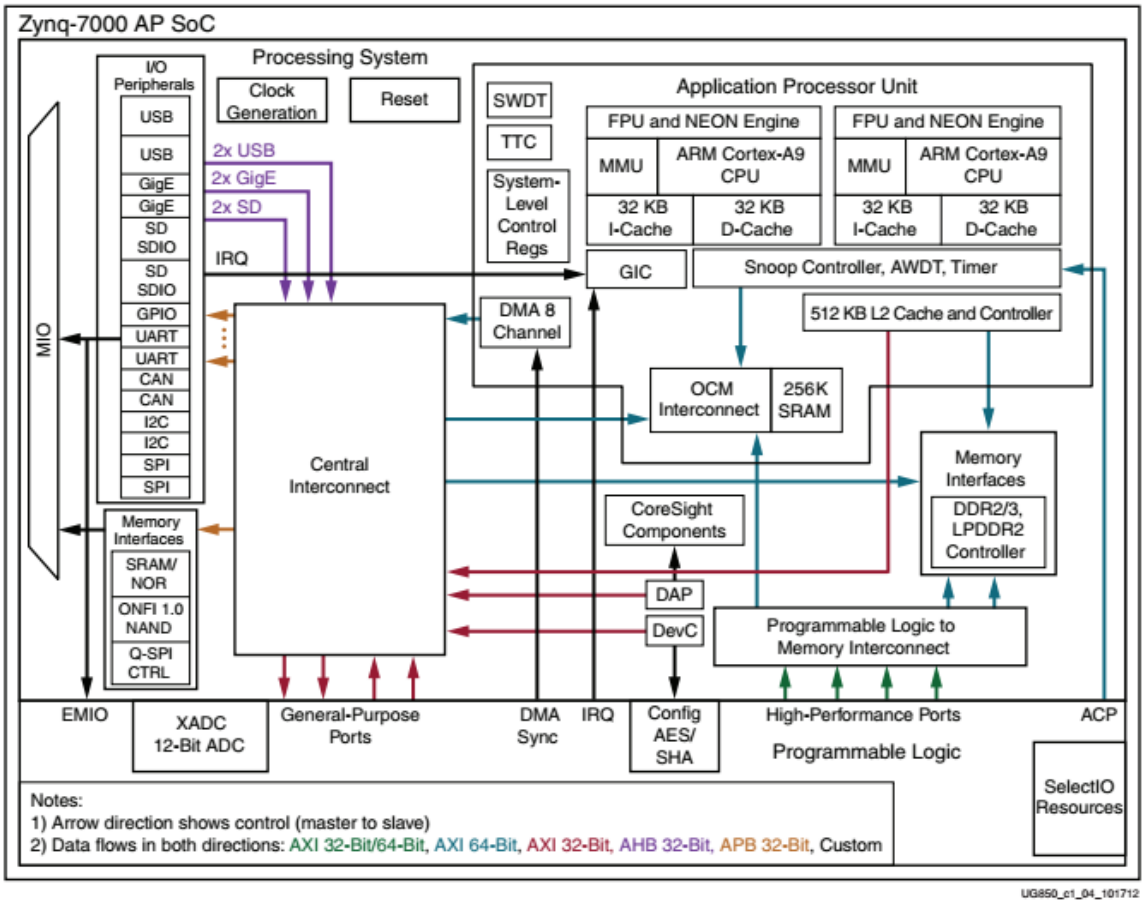


Εικόνα 35: Abstract αρχιτεκτονική Zynq

Όπως παρατηρούμε και στις δυο προηγούμενες εικόνες, το Zynq αποτελείται από δυο κύρια μέρη. Το Processing System (PS) και την Programmable Logic (PL). Το PS περιέχει τον ARM και μνήμη για τον ARM μεταξύ άλλων και η PL περιέχει τα συστατικά του παραδοσιακού FPGA. Η επικοινωνία τους γίνεται μέσω του AXI Bus το οποίο αποτελεί μέρος του AMBA Bus της ARM.

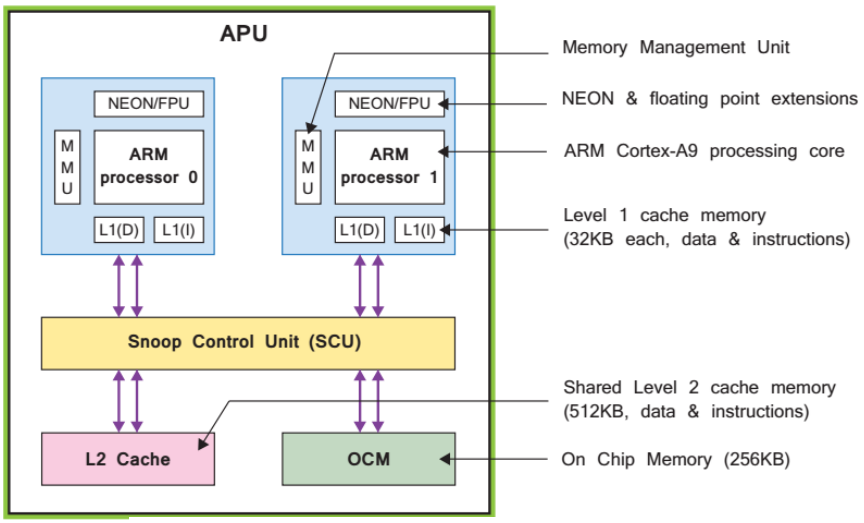
3.3.1 Σύστημα Επεξεργασίας (PS)

Όπως παρατηρούμε από το επόμενο σχήμα του Zynq, το PS οποίο αναφέρεται και ως Επεξεργαστική Μονάδα Εφαρμογής (Application Processor Unit – APU) αποτελείται εκτός από τον ARM και από διάφορα περιφερειακά Interfaces μνήμη cache, cache interfaces, κύκλωμα ρολογιού (μόνο της APU) καθώς και σύστημα διασύνδεσης και DMA controller.



Εικόνα 36: Αναλυτική δισύνδεση στο Zynq

Στην επόμενη εικόνα βλέπουμε ένα απλοποιημένο block diagram της μονάδας APU.



Εικόνα 37: APU Block Diagram

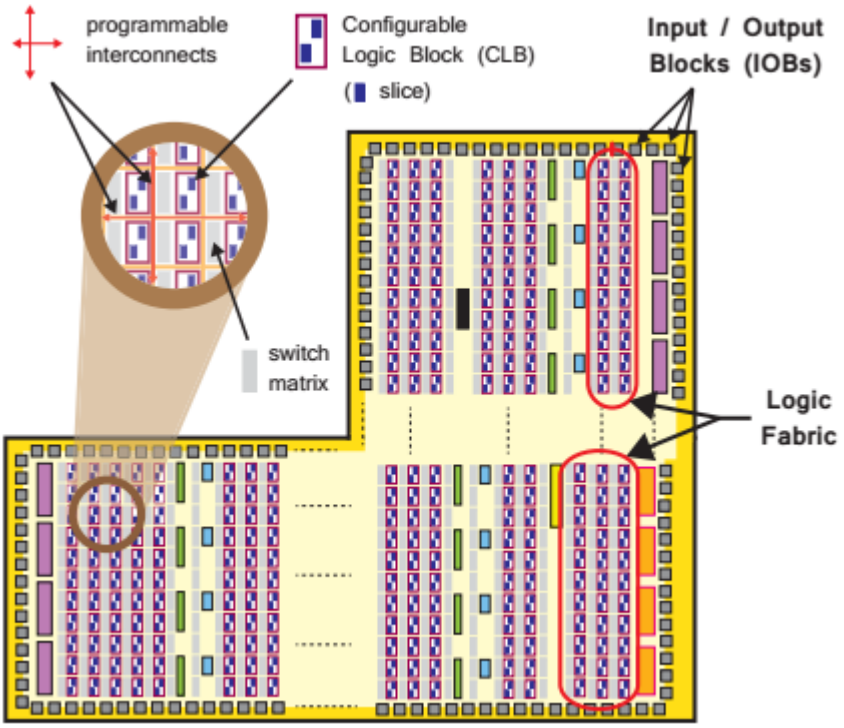
Ο Cortex A9 λειτουργεί μέχρι 1 GHz (ανάλογα τη συσκευή), έχουμε ξεχωριστές L1 caches κοινή L2 Cache και ως προς τα cores και ως προς το είδος των εντολών (data,instruction). Επίσης έχουμε ένα MMU (Memory Management Unit) καθώς και ένα Snoop Control Unit (SCU) για τη διασύνδεση μεταξύ της on chip memory (OCM) και της Cache με τα cores. Αναφορικά, η κύρια δουλειά της MMU είναι η μετάφραση μεταξύ φυσικής και ιδεατής μνήμης και του SCU συντονισμός μεταξύ L1, L2 καθώς και η διαιτησία (arbitration) για την προσπέλαση στην L2. Επίσης αναλαμβάνει και τον έλεγχο των μεταφορών μεταξύ της PL και του PS (δεν φαίνεται στο σχήμα) μέσω του interface ACP (Accelerator Coherency Port). Έχει μονάδες για την διαχείριση των διακοπών (interrupts) καθώς και timers. Επίσης οι επιταχυντές NEON παρέχουν SIMD (Single Instruction Multiple Data) λειτουργίες γεγονός που ευνοεί εφαρμογές όπως media και dsp. Η μονάδα FPU (Floating Point Unit) προσφέρει σημαντική αύξηση στην απόδοση μεταξύ FP αριθμών. Στην συνέχεια αναφέρουμε μερικές διασυνδέσεις (interfaces) του PS με εξωτερικές συσκευές.

I/O Interface	Description
SPI (x2)	Serial Peripheral Interface [10] *De facto standard for serial communications based on a 4-pin interface. Can be used either in master or slave mode.*
I2C (x2)	I^2C bus [14] *Compliant with the I2C bus specification, version 2. Supports master and slave modes.*
CAN (x2)	Controller Area Network *Bus interface controller compliant with ISO 118980-1, CAN 2.0A and CAN 2.0B standards.*
UART (x2)	Universal Asynchronous Receiver Transmitter *Low rate data modem interface for serial communication. Often used for Terminal connections to a host PC.*
GPIO	General Purpose Input/Output *There are 4 banks GPIO, each of 32 bits.*
SD (x2)	*For interfacing with SD card memory.*
USB (x2)	Universal Serial Bus *Compliant with USB 2.0, and can be used as a host, device, or flexibly ("on-the-go" or OTG mode, meaning that it can switch between host and device modes).*
GigE (x2)	Ethernet *Ethernet MAC peripheral, supporting 10Mbps, 100Mbps and 1Gbps modes.*

Εικόνα 38: Zynq I/O Interfaces

3.3.2 Προγραμματιζόμενη Λογική (PL)

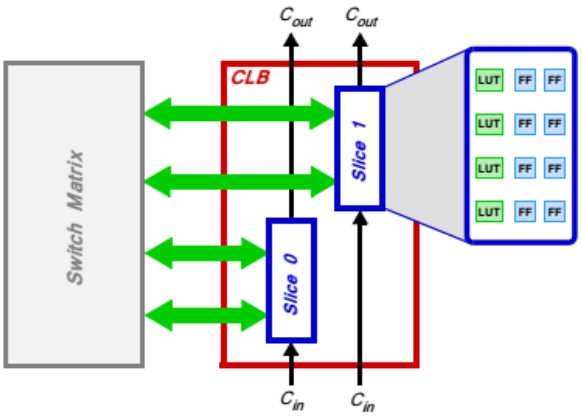
Η PL του Zynq είναι το «κλασικό» FPGA



Εικόνα 39: Progammable Logic του Zynq - CLB

Η PI βασίζεται ανάλογα με το Board στη σειρά Artix 7 ή Kintex 7 ανάλογα το μοντέλο. Και οι δυο σειρές μοιράζονται τα ακόλουθα στοιχεία:

- **Configurable Logic Block (CLB):** Αποτελεί το κύριο λειτουργικό στοιχείο ενός FPGA. Τα CLBs είναι μικρές ομαδοποιήσεις λογικών στοιχείων. Κάθε FPGA περιέχει ένα μεγάλο αριθμό από CLBs, τα οποία οργανώνονται σε μορφή δισδιάστατου πίνακα και συνδέονται με παρόμοιους πόρους μέσω (κάθετων και οριζόντιων) προγραμματιζόμενων διασυνδέσεων (programmable interconnects). Κάθε CLB περιέχει δύο slices και είναι τοποθετημένο δίπλα σε ένα πίνακα διακοπτών (switch matrix).
- **Slice:** Είναι μια υπο-μονάδα εντός ενός CLB. Περιέχει πόρους για την υλοποίηση συνδυαστικών και ακολουθιακών λογικών κυκλωμάτων. Όπως φαίνεται και στο επόμενο σχήμα, κάθε slice στο Zynq αποτελείται από 4 *Lookup Tables (LUTs),* 8 *Flip-Flop (FFs)* και άλλα λογικά στοιχεία.



Εικόνα 40: Σύνθεση ενός CLB

:

- **Switch Matrix:** Ένας πίνακας διακοπτών βρίσκεται δίπλα σε κάθε CLB. Παρέχει ευέλικτη υποδομή για τη δρομολόγηση συνδέσεων μεταξύ των στοιχείων ενός CLB, καθώς και μεταξύ του CLB και άλλων πόρων της PL.

- **Lookup Table (LUT):** Είναι ένας ευέλικτος πόρος, ικανός να υλοποιήσει:
  i. μια λογική συνάρτηση με έως και έξι εισόδους,
  ii. μια μικρή μνήμη μόνο-για-ανάγνωση (ROM),
  iii. μια μικρή μνήμη τυχαίας προσπέλασης (RAM),
  iv. έναν καταχωρητή ολίσθησης (shift register).

Οι LUTs μπορούν να συνδυαστούν μεταξύ τους για την υλοποίηση μεγαλύτερων λογικών συναρτήσεων, μνημών και καταχωρητών ολίσθησης.

- **Flip-Flop (FF):** Είναι ένα ακολουθιακό κυκλωματικό στοιχείο που υλοποιεί έναν καταχωρητή 1-bit, με λειτουργία επαναφοράς (reset). Προαιρετικά, ένα από τα FFs μπορεί να χρησιμοποιηθεί και ως μανδαλωτής (latch).

- **Carry Logic:** Τα αριθμητικά κυκλώματα απαιτούν τη μετάδοση ενδιάμεσων σημάτων μεταξύ γειτονικών slices, γεγονός που επιτυγχάνεται με την Carry Logic. Περιλαμβάνει μια αλυσίδα δρόμων και πολυπλεκτών για να συνδέσει slices σε μία κατακόρυφη στήλη.

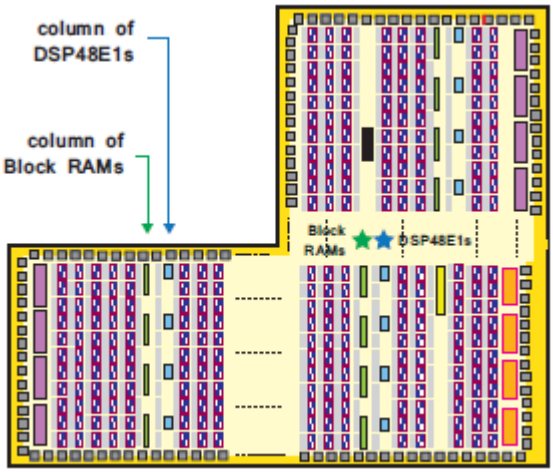
- **Input/Output Blocks (IOBs):** Παρέχουν διεπαφή μεταξύ των πόρων της προγραμματιζόμενης λογικής και των φυσικών ακροδεκτών που συνδέουν τη συσκευή με εξωτερικά κυκλώματα. Κάθε IOB μπορεί να διαχειριστεί ένα σήμα εισόδου ή εξόδου του 1 bit. Τα IOBs συνήθως βρίσκονται κοντά στην περίμετρο της συσκευής.

- **Block RAM:** Οι Block RAMs μπορούν να υλοποιήσουν μνήμες τυχαίας προσπέλασης (RAM), μνήμες μόνο-για-ανάγνωση (ROM) και καταχωρητές FIFO (First In First Out). Παράλληλα, υποστηρίζουν το πρωτόκολλο ECC (Error Correction Coding), το οποίο ανιχνεύει σφάλματα στα δεδομένα και μπορεί να διορθώσει σφάλματα του 1 Bit. Κάθε Block RAM μπορεί να αποθηκεύσει έως και 36Kb πληροφορίας και μπορεί να ρυθμιστεί είτε ως μία RAM των 36Kb είτε ως δύο ανεξάρτητες RAMs των 18Kb. Το default μέγεθος λέξης είναι 18 bits και με αυτή τη ρύθμιση κάθε Block RAM συνιστά 2048 στοιχεία μνήμης. Με αλλαγή στη ρύθμιση η Block RAM μπορεί να αναμορφωθεί ώστε να περιέχει είτε περισσότερα μικρά στοιχεία (π.χ. 4096 στοιχεία x 9 bits) είτε λιγότερα και μακρύτερα στοιχεία

(π.χ. 1024 στοιχεία x 36 bits, 512 x 72 bits). Μνήμες μεγαλύτερης χωρητικότητας μπορούν να σχεδιαστούν συνδυάζοντας δύο ή περισσότερες Block RAMs. Η ύπαρξη της Block RAM σημαίνει ότι μια μεγάλη ποσότητα δεδομένων μπορεί να αποθηκευτεί σε μικρό χώρο στη συσκευή, μέσα σε ένα βελτιστοποιημένο στοιχείο μνήμης. Η εναλλακτική λύση είναι η χρήση των LUTs του FPGA για τη δημιουργία μιας κατανεμημένης μνήμης (Distributed RAM). Όμως, για το σχηματισμό μνήμης ίδιου μεγέθους με μια Block RAM, απαιτείται σημαντικός αριθμός LUTs που εκτείνεται σε μεγαλύτερο χώρο, γεγονός που οδηγεί σε περισσότερες καθυστερήσεις κατά τη δρομολόγηση. Σε κάποιες περιπτώσεις, η χρήση LUTs έχει πλεονεκτήματα κυρίως λόγω της δυνατότητας ευέλικτης τοποθέτησης της μνήμης κοντά στα εκάστοτε στοιχεία του FPGA που τη χρησιμοποιούν.

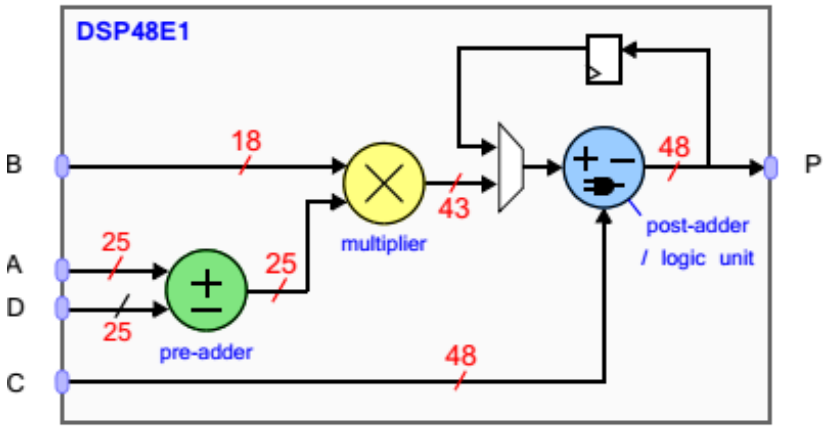
Στο Zynq- 7000 περιέχονται επιπλέον και κάποια στοιχεία ειδικού σκοπού τα *DSP48E1s* για αριθμητικές πράξεις υψηλής ταχύτητας.

Τοποθετούνται κοντά στις μνήμες και ο λόγος για αυτήν την τοποθέτηση είναι ότι οι εντατικοί υπολογισμοί και η αποθήκευση δεδομένων στη μνήμη είναι στενά συσχετισμένες λειτουργίες.



Εικόνα 41: Block Ram και DSP στην Pl του Zynq

DSP48E1s: Οι LUTs του FPGA μπορούν να χρησιμοποιηθούν για την υλοποίηση αριθμητικών τελεστών οποιουδήποτε μήκους, όμως είναι πιο κατάλληλοι για τελεστές με



Εικόνα 42: Αρχιτεκτονική DSP48E1

μικρά μήκη λέξης. Οι DSP48E1s είναι εξειδικευμένα στοιχεία για την υλοποίηση

αριθμητικών πράξεων υψηλής ταχύτητας σε σήματα με μέσο προς μεγάλο μήκος λέξης. Ένας DSP48E1s απαρτίζεται κυρίως από έναν προ-αθροιστή (με δυνατότητα πρόσθεσης και αφαίρεσης), έναν πολλαπλασιαστή και έναν ακόμα αθροιστή πριν την έξοδο, όπως φαίνεται στο επόμενο σχήμα, όπου αναγράφονται επίσης τα μέγιστα μήκη λέξης που μπορεί να διαχειριστεί κάθε υπομονάδα.

Ο DSP48E1 χρησιμοποιεί κυκλώματα πολυπλεξίας για την υποστήριξη δυναμικής αλλαγής των υπολογισμών (δηλ. αλλαγή των εκτελούμενων πράξεων σε κάθε κύκλο ρολογιού, αν αυτό χρειάζεται). Με κατάλληλη χρήση ενός, δύο, ή και όλων των υπομονάδων, είναι δυνατή η υλοποίηση πληθώρας αριθμητικών υπολογισμών. Επίσης, σημειώνεται ότι ο αθροιστής που βρίσκεται πριν την έξοδο, μπορεί, με ανάλογη ρύθμιση, να υλοποιήσει όλες τις βασικές λογικές πράξεις (NOT, AND, OR, NAND, NOR, XOR, XNOR). Με συνδυασμό πολλαπλών DSP48E1s μπορούν να υλοποιηθούν ακόμα πιο πολύπλοκοι υπολογισμοί και με μεγαλύτερο μήκος λέξεων. Παράλληλα, οι DSP48E1s μπορούν να χρονιστούν στη μέγιστη συχνότητα ρολογιού της συσκευής (όπως και οι Block RAMs) και έχουν χαμηλή κατανάλωση ισχύος. Αυτά τα χαρακτηριστικά, τους καθιστούν μια ελκυστική επιλογή για την υλοποίηση αριθμητικών κυκλωμάτων με υψηλές υπολογιστικές απαιτήσεις.

Εκτός από όσα αναφέραμε παραπάνω, στην προγραμματιζόμενη λογική του Zynq υπάρχουν και κάποιες ακόμα ειδικές εξωτερικές διεπαφές. Μερικές από αυτές είναι:

- **XADC Block:** Μονάδα μετατροπής αναλογικού σήματος σε ψηφιακό. Περιλαμβάνει δύο ξεχωριστούς ADCs (Analog-to-Digital Converters) των 12bit, ικανούς να δειγματοληπτήσουν εξωτερικά αναλογικά σήματα εισόδου σε 1 Msps.
- **Clocks:** Η προγραμματιζόμενη λογική λαμβάνει 4 ξεχωριστά σήματα ρολογιού από το σύστημα επεξεργασίας. Επιπρόσθετα, μπορεί να παράγει και να διαδίδει δικές της συχνότητες ρολογιού, ανεξάρτητα από το σύστημα επεξεργασίας.
- **JTAG:** Παρέχεται ένα σετ από θύρες JTAG για προγραμματισμό και αποσφαλμάτωση της προγραμματιζόμενης λογικής [12]. Η ρύθμιση μέσω JTAG χρησιμοποιείται συχνά κατά τη φάση ανάπτυξης της εφαρμογής. Παράλληλα, το JTAG υποστηρίζει την αποσφαλμάτωση με χρήση εργαλείων τόσο της ARM όσο και της Xilinx.

3.3.3 Επικοινωνία PS και PL

Είδαμε παραπάνω τις βασικές ιδιότητες των δύο μερών του Zynq-7000, του συστήματος επεξεργασίας (PS) και της προγραμματιζόμενης λογικής (PL). Ένα από τα πιο ελκυστικά χαρακτηριστικά του Zynq είναι η δυνατότητα της από κοινού χρήσης των δύο μερών, ώστε να προκύψει ένα πλήρες ολοκληρωμένο σύστημα. Για την επίτευξη αυτού του σκοπού υπάρχει ένα σύνολο *διασυνδέσεων (interconnects)* και *διεπαφών (interfaces)* πρωτοκόλλου AXI, το οποίο αποτελεί τη «γέφυρα» επικοινωνίας μεταξύ του PS και της PL.

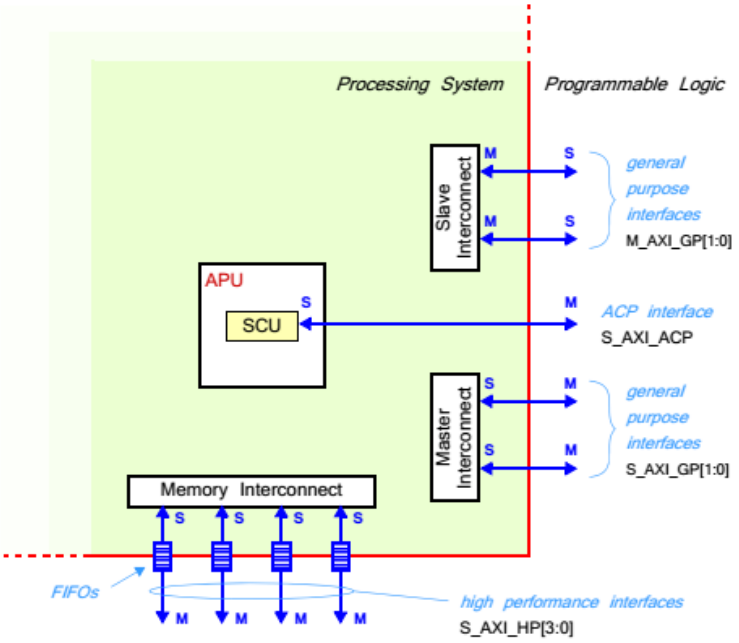
Τα αρχικά AXI αντιπροσωπεύουν το *Advanced eXtensible Interface*. Η τρέχουσα έκδοση είναι η AXI4, η οποία είναι μέρος του ανοιχτού προτύπου ARM AMBA 3.0 (πρότυπο on-chip επικοινωνίας). Οι δίαυλοι AXI μπορούν να χρησιμοποιηθούν ευέλικτα και σε γενικές γραμμές συνδέουν τον επεξεργαστή με άλλα τμήματα του υλικού σε ένα ενσωματωμένο σύστημα. Υπάρχουν 3 κατηγορίες AXI4, καθεμία από τις οποίες αντιπροσωπεύει ένα διαφορετικό πρωτόκολλο διαύλου:

- **AXI4:** Χρησιμοποιείται για memory-mapped (αναφέρονται σε συγκεκριμένη θέση μνήμης) συνδέσμους και προσφέρει μέγιστη απόδοση: Παρέχεται μια διεύθυνση και ακολουθεί η μεταφορά μιας ριπής δεδομένων μεγέθους έως και 256 λέξεις.
- **AXI4-Lite:** Ένας απλοποιημένος σύνδεσμος, που υποστηρίζει μόνο μια μεταφορά δεδομένων ανά σύνδεση (χωρίς ριπές). Είναι επίσης memory-

mapped: Σε αυτήν την περίπτωση μεταφέρεται μια διεύθυνση και μόνο μια λέξη δεδομένων.

- **AXI4-Stream:** Χρησιμοποιείται για υψηλής ταχύτητας ροή δεδομένων. Στο AXI4-Stream δεν υπάρχει μηχανισμός για συγκεκριμένες διευθύνσεις μνήμης (non memory-mapped επικοινωνία). Είναι κατάλληλο για άμεση ροή δεδομένων μεταξύ της πηγής και του προορισμού.

Η κύρια διεπαφή μεταξύ του συστήματος επεξεργασίας και της προγραμματιζόμενης λογικής απαρτίζεται από ένα σετ 9 AXI διεπαφών, από τις οποίες καθεμία έχει πολλαπλά



Εικόνα 43: Interfaces PS/PI

κανάλια. Αυτές οι διεπαφές κάνουν ειδικές συνδέσεις ανάμεσα στην PL και σε διασυνδέσεις (interconnects) εντός του PS.

Για λόγους σαφήνειας, σημειώνουμε τους ορισμούς αυτών των όρων:

- **Διασύνδεση (Interconnect)** — Είναι ουσιαστικά ένας διακόπτης, ο οποίος διαχειρίζεται και κατευθύνει την κίνηση μεταξύ AXI διεπαφών. Υπάρχουν αρκετές τέτοιες διασυνδέσεις μέσα στο PS. Κάποιες από αυτές έχουν άμεση διεπαφή με την PL και άλλες είναι μόνο για εσωτερική χρήση. Οι συνδέσεις ανάμεσα σε αυτές τις διασυνδέσεις γίνονται και πάλι μέσω AXI διεπαφών.
- **Διεπαφή (Interface)** — Είναι μια σύνδεση από σημείο σε σημείο (point-to- point) για τη μεταφορά δεδομένων, διευθύνσεων και σημάτων χειραψίας ανάμεσα σε masters και slaves εντός του συστήματος.

Παρατηρούμε ότι όλες οι διεπαφές είναι συγκεκριμένα συνδεδεμένες με διασυνδέσεις εντός του PS, με την εξαίρεση της διεπαφής ACP που συνδέεται απευθείας στο Snoop Control Unit της APU, την οποία θα εξηγήσουμε και παρακάτω.

Βλέπουμε επίσης ότι οι διασυνδέσεις μέσα στο PS που συνδέονται με την PL είναι τρεις: *Memory Interconnect, Master Interconnect* και *Slave Interconnect.* Στο επόμενο σχήμα. παρατίθεται μια λίστα με όλες τις διεπαφές PS-PL, οι οποίες φαίνονται με μπλε βέλη και στο προηγούμενο.

Πίνακας 1: Διεπαφές μεταξύ PS και PI

Interface Name	Interface Description	Master	Slave
M_AXI_GP0	General Purpose (AXI_GP)	PS	PL
M_AXI_GP1		PS	PL
S_AXI_GP0	General Purpose (AXI_GP)	PL	PS
S_AXI_GP1		PL	PS
S_AXI_ACP	Accelerator Coherency Port (ACP), cache coherent transaction	PL	PS
S_AXI_HP0	High Performance Ports (AXI_HP) with read/write FIFOs. (Note that AXI_HP interfaces are sometimes referred to as AXI Fifo Interfaces, or AFIs).	PL	PS
S_AXI_HP1		PL	PS
S_AXI_HP2		PL	PS
S_AXI_HP3		PL	PS

Σημειώνεται μια μικρή περιγραφή της διεπαφής και η κατανομή των ρόλων του master και του slave (ο master είναι αυτός που ελέγχει το δίαυλο επικοινωνίας και κινεί τη συναλλαγή, ενώ ο slave αποκρίνεται στις εντολές του master).

Όπως βλέπουμε, υπάρχουν τρεις κατηγορίες διεπαφών. Πιο αναλυτικά:

- **General Purpose AXI** — Δίαυλος δεδομένων των 32bit, ο οποίος είναι κατάλληλος για επικοινωνίες χαμηλού και μεσαίου ρυθμού μεταξύ του PS και της PL. Αυτή η διεπαφή είναι άμεση, δηλαδή τα δεδομένα μεταφέρονται κατευθείαν χωρίς κάποιο ενδιάμεσο στάδιο αποθήκευσης (buffering). Υπάρχουν τέσσερις τέτοιες διεπαφές, από τις οποίες στις δύο είναι master το σύστημα επεξεργασίας και στις άλλες δύο είναι master η προγραμματιζόμενη λογική.
- **Accelerator Coherency Port (ACP)** — Διεπαφή που αναφέρθηκε και παραπάνω και αποτελεί τη μόνη ασύγχρονη σύνδεση ανάμεσα στην PL και το Snoop Control Unit της APU. Ο δίαυλος διαθέτει πλάτος 64bit. Αυτή η διεπαφή χρησιμοποιείται για την επίτευξη συνοχής ανάμεσα στις μνήμες cache της APU και στα στοιχεία της PL. ο master της επικοινωνίας είναι η προγραμματιζόμενη λογική.
- **High Performance Ports** — Οι τέσσερις AXI διεπαφές υψηλής απόδοσης συνδέουν την PL με την Memory Interconnect του PS και περιλαμβάνουν FIFO buffers για να γίνεται δυνατή η διαδικασία ανάγνωσης και εγγραφής δεδομένων «κατά ριπάς». Υποστηρίζουν υψηλού ρυθμού επικοινωνία ανάμεσα στην προγραμματιζόμενη λογική και σε στοιχεία μνήμης στο σύστημα επεξεργασίας. Το μέγεθος διαύλου είναι είτε 32 είτε 64 bits και η PL είναι ο master και των τεσσάρων διεπαφών.

Αυτά ήταν τα βασικά κομμάτια διασυνδέσεων και υλικού στο Zynq. Αξίζει όμως να γίνει μια αναφορά και σε ένα άλλο χαρακτηριστικό του Zynq που έχει αποκτήσει ιδιαίτερη σημασία τα τελευταία χρόνια και είναι η ασφάλεια. Καθότι τα FPGAs χρησιμοποιούνται όλο και περισσότερο ακόμα και για εμπορικούς σκοπούς, υπάρχουν όλο και περισσότερες εφαρμογές τους (αεροναυπηγική, αυτοκινητοβιομηχανία κλπ) όπου το υλικό ή/και το λογισμικό θέλουν να είναι όσο το δυνατόν πιο προστατευμένο. Έτσι η Xilinx χρησιμοποιεί ένα σύνολο μεθόδων προς αυτήν την κατεύθυνση.

Secure Boot: Το Zynq σε κάθε Boot ή reset bootάρει αποκλειστικά από την sd card (ή γενικά από εξωτερική μνήμη) και στη συνέχει προγραμματίζει την PL. Με αυτόν τον τρόπο εξασφαλίζεται ότι δεν μπορεί να φορτωθεί ούτε κάποιο επικίνδυνο bitstream στην PL ούτε κάποιο επικίνδυνο software στον ARM. Επίσης ενσωματώνονται αρκετές λειτουργίες

encrypted bootloader και κλειδιών κρυπτογράφησης έτσι ώστε μόνο όποιος έχει τα ιδιωτικά κλειδιά να μπορεί να προγραμματίσει το FPGA.

Παρακάτω αναφέρονται ένα σύνολο πρακτικών ασφαλείας που έχουν υποστήριξη σε Hardware υλοποίηση:

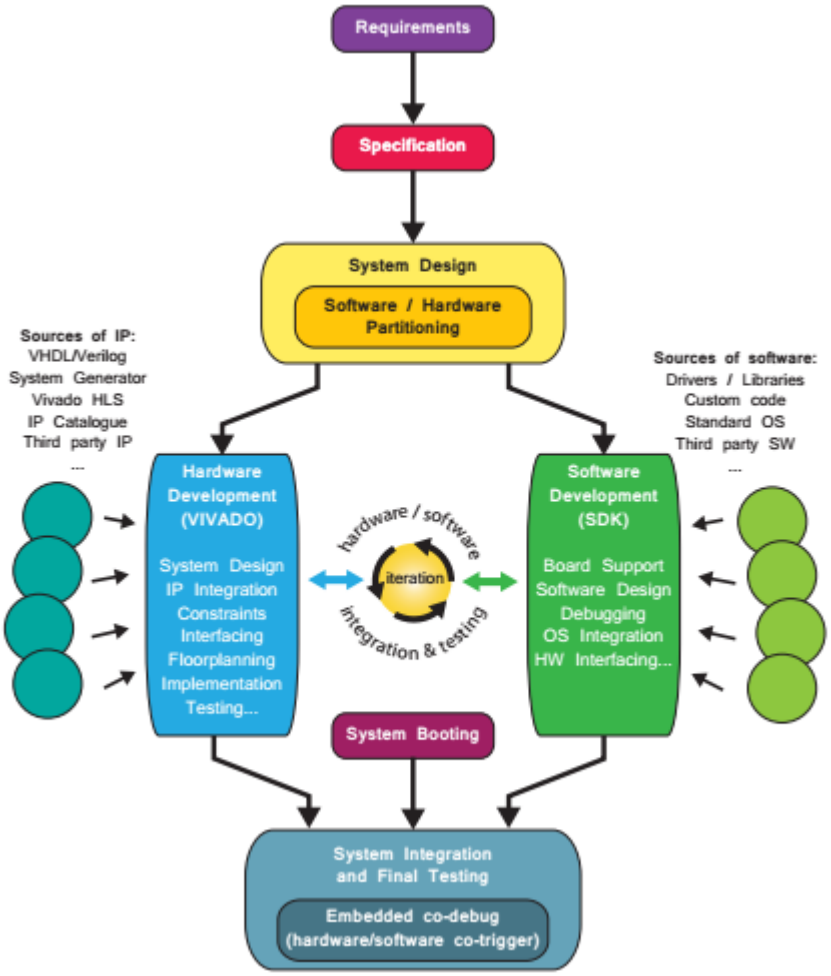
- ARM TrustZone support (PS and PL)
- AES-256 encryption (BBRAM key and eFUSE key)
- Secure Configuration and Boot (PS and PL)
- HMAC bitstream authentication
- FSBL RSA-2048 Authentication
- Hardened readback disable
- JTAG disable/monitor
- SEU checker

Runtime support: Προσφέρει διάφορους τρόπους για να προστατευθεί το σύστημα είτε πρόκειται για την PL είτε για τη PS ενώ το σύστημα λειτουργεί (runtime).

ARM TrustZone Technology: Κάποια κομμάτια του hardware είναι διαθέσιμα σε όλες τις εφαρμογές αλλά μπορούν να είναι διαθέσιμα μόνο σε ασφαλείς εφαρμογές. Με αυτόν τον τρόπο αποφεύγεται η περίπτωση μια μη ασφαλής επιλογή να έχει πρόσβαση σε σημείο που είναι «ασφαλές» ενώ μπορούν να λειτουργούν και να επικοινωνούν και τα 2 μεταξύ τους (ασφαλή και μη) ταυτόχρονα.

3.4 Προγραμματισμός του FPGA

Ο Προγραμματισμός του FPGA είναι γενικά μια σχετικά περίπλοκη διαδικασία, που περιέχει ένα σύνολο από αρκετά πολύπλοκες διαδικασίες οι οποίες συνήθως δεν είναι εμφανείς στο χρήστη. Η πολυπλοκότητα όπως είναι λογικό αυξάνει αρκετά στο Zynq εφόσον έχουν να προγραμματιστούν δυο διαφορετικά κομμάτια και μετά αυτά πρέπει να επικοινωνήσουν μεταξύ τους σωστά. Παρακάτω βλέπουμε ένα πολύ αφαιρετικό μοντέλο της διαδικασίας που πρέπει να ακολουθηθεί.



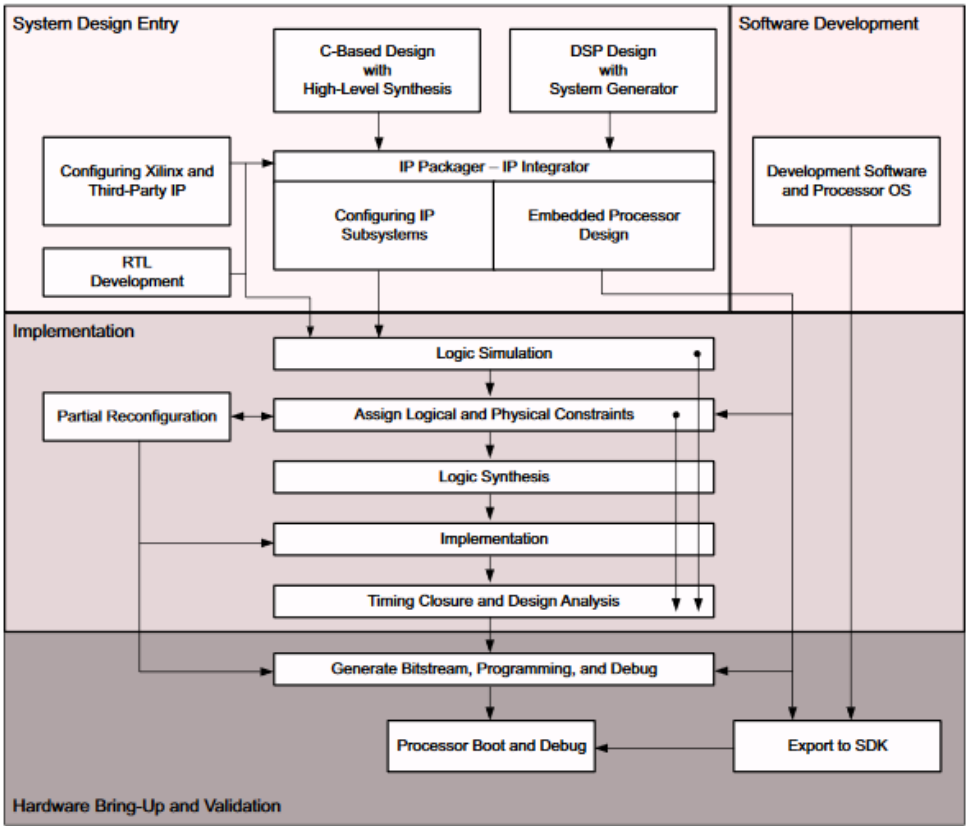
Εικόνα 44: Αφαιρετικό μοντέλο προγραμματισμού Zynq

3.5 Vivado Design Suite (VDS)

Η Xilinx παρέχει μια σουίτα εφαρμογών με σκοπό τον προγραμματισμό του FPGA [12]. Τα βασικά εργαλεία αυτής είναι τα:

1. Vivado HLS
2. Vivado
3. Xilinix Software Development Kit (SDK)

Κάθε ένα εκτελεί διαφορετική εργασία και συμβάλει στην υλοποίηση του παρακάτω Design Flow



Εικόνα 45: Vivado Design Suite Design Flow

3.5.1 Vivado High-Level Synthesis (HLS)

Σκοπός της διπλωματικής είναι, όπως αναφέρθηκε προηγουμένως, η εξερεύνηση των δυνατοτήτων υλοποίησης αλγορίθμων στο υλικό μέσω του High-Level Programming. Το εργαλείο που η Xilinx έχει για τη συγκεκριμένη εργασία ονομάζεται Vivado HLS και είναι δωρεάν με την υπόλοιπη σουίτα της εταιρίας που ονομάζεται Vivado Design Suite (VDS). Το εργαλείο βασίζεται στο λογισμικό ανάπτυξης κώδικα Eclipse το οποίο είναι δωρεάν λογισμικό και πολύ δημοφιλές, παρέχοντας μια ολοκληρωμένη σουίτα ανάπτυξης, με πολύ φιλικός προς τον χρήστη interface.

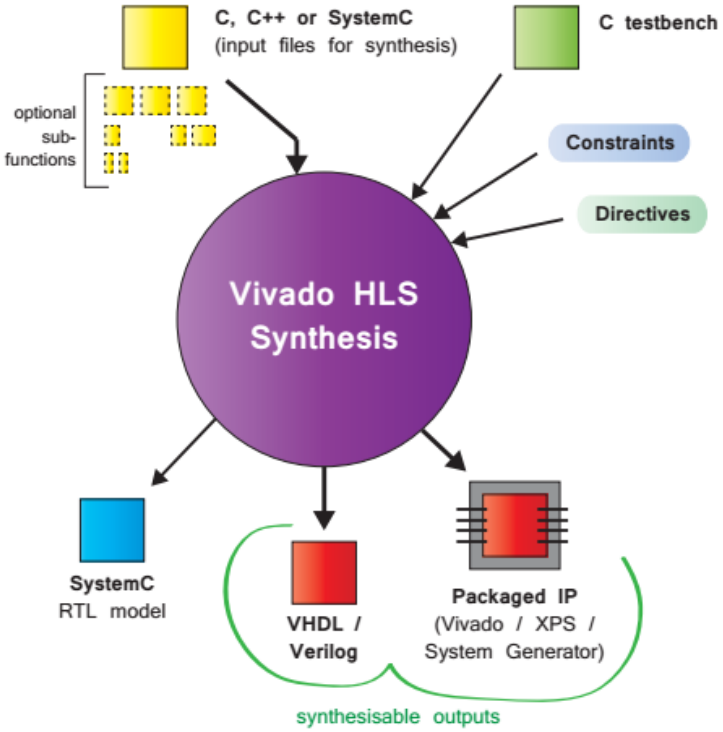
Ως εργαλείο δέχεται ως είσοδο:

1. **Αρχεία σε γλώσσες** C, C++ και SystemC τα οποία περιγράφουν το hardware που θέλουμε να κατασκευάσουμε (πχ rasterizer, interpolation κλπ)
2. **C Αρχεία για το Testbench**. Τα αρχεία αυτά περιγράφουν το test που θέλουμε να εκτελέσουμε στο hardware
3. **Constraints (Tcl):** Τα Constraints αυτά χρησιμοποιούνται ώστε να καθορίσουμε την επιθυμητή συσκευή, περίοδο ρολογιού καθώς και χρόνο αβεβαιότητας του ρολογιού (clock uncertenaty figures)
4. **Directives (Ντιρεκτίβες) (Tcl):** Αφορούν διάφορες «οδηγίες» ως προς το εργαλείο σχετικά με το πως θέλουμε να πραγματοποιήσει τον κώδικα που του περιγράφουμε πχ Pipeline, datapack κλπ

Οι έξοδοι του προγράμματος είναι:

1. Μοντέλο SystemC: Ένα μοντέλο SystemC σχεδιασμένο για RTL simulation
2. RTL Code: Κώδικας σε VHDL και Verilog ο οποίος μπορείς να γίνει simulation ή να γίνει σύνθεση σε **κάποιο Xilinx FPGA**
3. Packaged IP το οποίο είναι ένα πακέτο με βάση το πρότυπο IP-XACT Και χρησιμοποιείται για εύκολη εισαγωγή ως IP σε κάποιο άλλο Project

Μια σχηματική απεικόνιση των όσων περιγράψαμε:



Εικόνα 46: Μοντέλο Παραγωγής RTL μέσω Vivado HLS

Το Vivado HLS υποστηρίζει τα περισσότερα στοιχεία και βιβλιοθήκες της C. Υπάρχουν όμως και κάποιοι περιορισμοί, όπως και κάποια στοιχεία τα οποία επεκτείνουν την C ώστε το hardware που θα δημιουργείται να είναι πιο αποδοτικό (efficient) [13].

Κλασικοί τύποι δεδομένων C-C++:

Πίνακας 2: Data types στην C

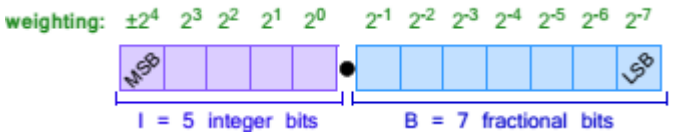
Type	Description	Number of Bits[a]	Range[b]
`char`	Representation of the basic character set.	8	-128 to 127
`signed char`		8	-128 to 127
`unsigned char`		8	0 to 255
`short int`	A reduced precision version of `int`, requiring less storage.	16	-32,768 to 32,767
`unsigned short int`		16	0 to 65,535
`int`	The basic integer data type.	32	-2,147,483,648 to 2,147,483,647
`unsigned int`		32	0 to 4,294,967,295
`long int`	In many cases the `long int` type will be the same length as `int`, i.e. 32 bits.	32	-2,147,483,648 to 2,147,483,647
`unsigned long int`		32	0 to 4,294,967,295
`long long int`	An extended precision integer type.	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long int`		64	0 to 18,446,744,073,709,551,615
`float`	Single precision floating point (IEEE 754)	32	$-3.403e^{+38}$ to $3.403e^{+38}$
`double`	Double precision floating point (IEEE 754)	64	$-1.798e^{+308}$ to $1.798e^{+308}$

Επιπλέον των προαναφερθέντων τύπων, το εργαλείο προσφέρει βιβλιοθήκη προσαρμοσμένου μήκους δεδομένων (arbitrary precision)

Πίνακας 3: Arbitary types στο Vivado HLS

Language	Integer Data Type	Description	Required Header
C	int*N* (e.g. `int7`)	signed integer of *N* bits precision	`#include "ap_cint.h"`
	uint*N* (e.g. `uint7`)	unsigned integer of *N* bits precision	
C++	ap_int<*N*> (e.g. `ap_int<7>`)	signed integer of *N* bits precision	`#include "ap_int.h"`
	ap_uint<*N*> (e.g. `ap_uint<7>`)	unsigned integer of *N* bits precision	

Επίσης προσφέρονται τύποι για δεδομένα σταθερής υποδιαστολής (fixed point)



Εικόνα 47: Fixed point number

Πίνακας 4: Fixed point numbers στο HLS

Language	Fixed Point Data Type	Description	Required Header
C++	ap_fixed<*W,I,Q,O,N*>	Signed fixed point number of *I* integer bits and *W-I* fractional bits.	`#include "ap_fixed.h"`
	ap_ufixed<*W,I,Q,O,N*>	Unsigned fixed point number of *I* integer bits and *W-I* fractional bits.	

Καθότι, όπως ανέφερα, υποστηρίζονται 3 γλώσσες έχουμε διαφορετικά παραδείγματα και περιπτώσεις για την κάθε μία. Εγώ αναφέρω μόνο της C/C++ που χρησιμοποίησα για την εκπόνηση της διπλωματικής

Μία αναφορά πρέπει να γίνει στο interface των blocks και των σημάτων. Στο software όταν σχεδιάζουμε μια συνάρτηση ορίζουμε εισόδους και εξόδους. Όταν αυτό μεταφράζεται σε hardware τότε αυτό μεταφράζεται σε κάποιο interface. Ανάλογα τον τύπο εισόδου/ εξόδου κλπ το HLS καθορίζει και τον τύπο που θα εκχωρήσει σε κάθε ένα από αυτά. Τα πρωτόκολλα διεπαφής είναι:

- ***ap_none*** — Το πιο απλό προτόκολο. Απλά ένα pin χώρις κάποιον extra συγχρονισμό
- ***ap_stable*** — Παρόμοιο με το ap_none με τη διαφορά ότι προορίζεται για δεδομένα που αλλάζουν σπάνια (συνήθως μετά από το reset). Προορίζονται για περιπτώσεις όπως οι configuration registers για παράδειγμα. Αφορά μόνο εισόδους και όχι εξόδους.
- ***ap_ack*** — Σε αυτήν την περίπτωση, αν το σήμα είναι είσοδος, προστίθεται ένα σήμα εξόδου που γίνεται 1 όταν διαβαστεί η είσοδος (άμεσα). Στην περίπτωση που είναι έξοδος εισάγει ένα σήμα εισόδου ότι η έξοδος διαβάστηκε ώστε να προχωρήσει στην επόμενη
- ***ap_vld*** — Στην περίπτωση εισόδου εισάγει ένα σήμα εισόδου το οποίο αν είναι ένα σημαίνει ότι τα δεδομένα της εισόδου της θύρας είναι valid. Στην περίπτωση που είναι έξοδος, εισάγει ένα σήμα εξόδου που ειδοποιεί πότε είναι valid τα δεδομένα
- ***ap_ovld*** — Παρόμοιο με το προηγούμενο, με τη διαφορά ότι μπορεί να εφαρμοστεί μόνο σε έξοδο ή σε inout (αμφίδρομη) και να αφορά τα δεδομένα εξόδου.
- ***ap_hs*** — Αποτελεί ένα υπερσύνολο των προαναφερθέντων προτκόλλων και υλοποιείται σε εισόδους και εξόδους. Το __hs αναφέρεται στο όνομα handshaking (χειραψία) δηλαδή αναλαμβάνει την διασφάλιση της σωστής επικοινωνίας μεταξύ των δυο μερών (valid, read). Χρειάζεται δύο επιπλέον σήματα.

- ***ap_memory*** — Αυτό αφορά μόνο πίνακες οι οποίοι χρησιμοποιούνται για να υλοποιήσουν μια μνήμη. Προσφέρει τη δυνατότητα μεταφορών τυχαίας

προσπέλασης Υποστηρίζει εισόδους, εξόδους και αμφίδρομες θύρες και χρειάζεται σήματα για ρολόι, για write enable καθώς και για διεύθυνση μνήμης.

- ***bram*** — Το ίδιο με το προηγούμενο με τη διαφορά ότι όταν χρησιμοποιούνται από τον IP Integrator ομαδοποιούνται σε μία κοινή θύρα.
- ***ap_fifo*** — Είναι συμβατό με πίνακες όπως και το προηγούμενο, με την διαφορά ότι η προσπέλαση των δεδομένων γίνεται σειριακά και όχι με τυχαία σειρά. Το γεγονός αυτό καθιστά την υλοποίησή της ευκολότερη (δεν χρειάζεται διεύθυνση). Ενσωματώνει σήματα ελέγχου ώστε να αποφευχθεί υπερχείλιση (overflow) ή υποχείληση (underflow)
- ***ap_bus*** — Αποτελεί μια διεπαφή γενικού διαύλου και δεν συνδέεται με κάποιο συγκεκριμένο standard. Στη συνέχεια μπορεί να ενσωματωθεί με τον δίαυλο συστήματος. Υποστηρίζει απλές αναγνώσεις και εγγραφές (single reads and writes) αλλά και συνεχόμενες μεταφορές (burst transfers). Αυτά καθορίζονται από ένα σύνολο σημάτων ελέγχου. Χρησιμοποιώντας κάποια ντιρεκτίβα μπορεί να καθοριστεί η χρήση του AXI κατά τη σύνθεση.
- ***axis*** — Καθορίζει τη χρήση AXI stream πρωτοκόλλου.
- ***s_axilite*** — Καθορίζει τη χρήση AXI Slave Lite πρωτοκόλλου.
- ***m_axi*** — Καθορίζει τη χρήση AXI Master πρωτοκόλλου

Στον παρακάτω πίνακα βλέπουμε τα πρωτόκολλα που υποστηρίζονται για κάθε τύπο εισόδου εξόδου και τις προκαθορισμένες (default) ρυθμίσεις του HLS.

Πίνακας 5: Πρωτόκολλα διεπεφής στο HLS

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	pass-by-value			pass-by-reference			pass-by-reference			pass-by-reference		
Interface Type[a]	I	IO	O	I	IO	O	I	IO	O	I	IO	O
ap_none	D	-	-	D	S	S	-	-	-	D	S	S
ap_stable	S	-	-	S	S	-	-	-	-	S	S	-
ap_ack	S	-	-	S	S	S	-	-	-	S	S	S
ap_vld	S	-	-	S	S	D	-	-	-	S	S	D
ap_ovld	-	-	-	-	D	S	-	-	-	-	D	S
ap_hs	S	-	-	S	S	S	S	-	S	S	S	S
ap_memory	-	-	-	-	-	-	D	D	D	-	-	-
bram	-	-	-	-	-	-	S	S	S	-	-	-
ap_fifo	-	-	-	S	-	S	S	-	S	S	-	S
ap_bus	-	-	-	S	S	S	S	S	S	S	S	S
axis	S	-	-	S	-	S	S	-	S	S	-	S
s_axilite	S	-	S	S	S	S	-	-	-	S	S	S
m_axi	-	-	-	S	S	S	S	S	S	S	S	S

Τα προαναφερθέντα πρωτόκολλα αφορούν το κάθε στοιχείο ενός block μιας συνάρτησης ξεχωριστά. Όμως υπάρχουν πρωτόκολλα διασύνδεσης που αφορούν ολόκληρο το block (συνάρτηση του προγράμματος) και χρησιμοποιούνται για το συντονισμό της επικοινωνίας τους. Ονομάζονται Block-Level protocols και είναι τα εξής:

- ***ap_ctrl_none*** — Είναι το απλούστερο πρωτόκολλο και δεν παρέχει κανένα σήμα συγχρονισμού. Ο συγχρονισμός θα πρέπει να επιτευχεθί χρησιμοποιώντας μόνο τα σήματα ελέγχου των επιμέρους θυρών
- ***ap_ctrl_hs*** —Στο συγκεκριμένο πρωτόκολλο παράγονται τα σήματα ελέγχου ap_ready, ap_idle και ap_done. Πιο συγκεκριμένα το ap_ready γίνεται 1 όταν το block είναι έτοιμο να δεχθεί νέες εισόδους, το ap_idle όντας ανενεργό σημαίνει ότι το block εκτελεί κάποια επεξεργασία ανώ το ap_done γίνεται ενεργό όταν τα όλα δεδομένα εξόδου έχουν πάρει τη σωστή τιμή. Χρησιμεύει ιδιαίτερα στην περίπτωση που θέλουμε να συνδέσουμε κάποιο απλό HLS block ή το top level block με κάποιο εξωτερικό επεξεργαστή ή module αφού ο έλεγχος και ο συγχρονισμός είναι αρκετά απλός.
- ***ap_ctrl_chain*** —Το συγκεκριμένο πρωτόκολλο είναι παρόμοιο με το προηγούμενο, με τη διαφορά ότι εισάγεται ένα καινούριο σήμα ap_continue. Αυτό χρησιμεύει στη διασύνδεση μιας αλυσίδας από HLS blocks. Το ap_continue όταν είναι ενεργό δείχνει τη δυνατότητα να δεχθεί καινούρια δεδομένα. Όταν γίνει ανενεργό σημαίνει ότι δε είναι έτοιμο για νέα δεδομένα και ότι ο αποστολέας θα πρέπει να περιμένει να γίνει ενεργό πριν την αποστολή νέων δεδομένων.

Σε κάθε block υπάρουν ως είσοδοι τα σήματα ap_clk που αφορούν το ρολόι του block και το ap_rst που αφορά το reset. Μπορούν επίσης,να συνδυαστούν και σήματα του block και των ports «φτιάχνοντας» ένα AXI-4 light interface.

Όπως εξηγήσαμε στο HLS μπορούμε να εισάγουμε «οδηγίες» προς το εργαλείο ώστε να «ελέγξουμε» τον τρόπο που θα υλοποιηθεί κάθε ένα από τα πράγματα που περιγράφουμε. Συγκεκριμένα στα πρωτόκολλα έχουμε τις εξής επιλογές ντιρεκτίβων:

- ***Array Map*** — Αφορά το συνδυασμό ενός συνόλου πινάκων με σκοπό να χρησιμοποιηθούν λιγότερα Ram blocks ή και FIFOs οδηγώντας αντίστοιχα και σε μείωση των σημάτων ελέγχου.
- ***Array Partition*** — Είναι το αντίστροφο της προαναφερθείσας ντιρεκτίβας και αφορά στη διαίρεση ενός array interface σε περισσότερα κομμάτια, οδηγώντας σε αύξηση των πόρων και των σημάτων ελέγχου, αλλά συγχρόνως και σε αύξηση του bandwidth.
- ***Array Reshape*** — Χρησιμοποιείται για την αναδιαμόρφωση στη μνήμη ενός πίνακα σε μικρότερους και στη συνέχεια την «ένωσή» τους σε έναν πιο πλατύ πίνακα με λιγότερες όμως διευθύνσεις. Αυτό οδηγεί σε λιγότερες θέσεις μνήμης και άρα μικρότερο μέγεθος διεθυνσιοδότησης
- ***Interface*** — Καθορίζει το πρωτόκολλο του block σε περίπτωση που το θέλουμε διαφορετικό πρωτόκολλο από το default του HLS. Όπως αναφέρθηκε μπορεί να είναι *ap_none*, *ap_ctrl_block* ή *ap_ctrl_chain*,
- ***Resource*** — Καθορίζει το hardware που θα υλοποιηθεί μια θύρα. Παράδειγμα μπορούμε να καθορίσουμε για ένα ap_memory interface να χρησιμοποιήσει dual port μνήμη αντί single port ή μια Fifo να υλοποιηθεί με Block RAM ή LUTs αντίστοιχα.
- ***Stream*** — Ορίζοντας μια θύρα ως streaming ορίζουμε την υλοποίησή της με FIFO κατά την οποία μπορούμε να επιλέξουμε και το μέγεθός της.

Όπως καταλαβαίνουμε, ακόμα και για το ίδιο κομμάτι κώδικα, μπορεί να «παραχθεί» αρκετά διαφορετικός κώδικας σε επίπεδο RTL που θα περιγράφει το διαφορετικό hardware που θα χρησιμοποιηθεί.

Τα στάδια που ακολουθούνται σε μια διαδικασία υλοποίησης με το HLS είναι τα εξής:

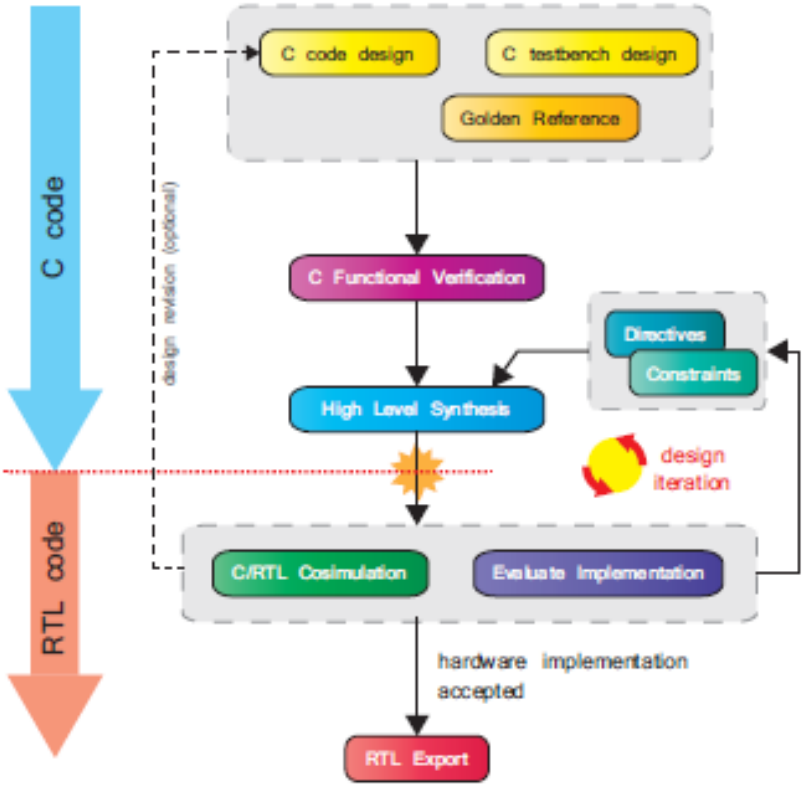
1. **Software simulation**: Κατά την ανάπτυξη του προγράμματός μας, ελέγχουμε με ένα testbench τη λειτουργία που επιθυμούμε. Αφορά απλά μια μεταγλώττιση (compile) του προγράμματός μας με κάποιον compiler. Η μόνη διαφορά σε σχέση με άλλα προγράμματα ανάπτυξης κώδικα είναι ότι το testbench πρέπει αναγκαστικά να είναι σε ξεχωριστό αρχείο σε σχέση με το κεντρικό πρόγραμμα. Αν αυτό το στάδιο επιτευχθεί, τότε πηγαίνουμε στο επόμενο στάδιο:
2. **C/RTL co-simulation**: Σε αυτό το στάδιο το HLS αναλαμβάνει να κάνει σύνθεση το πρόγραμμά μας και στη συνέχεια να εκτελέσει το ίδιο test. Εκτός του ότι προφανώς ο χρόνος που διαρκεί η σύνθεση εξαρτάται από το πρόγραμμα που έχει να «μεταφράσει». Όταν αυτό επιτευχθεί, το co-simulation πλέον διαρκεί τάξεις μεγέθους περισσότερο χρόνο. Σε αυτό το στάδιο βέβαια αποκτούμε μια ακριβή εικόνα του hardware που χρησιμοποιήθηκε καθώς μπορούμε να δούμε και αναλυτικά τις τιμές των σημάτων σε κάθε κύκλο ρολογιού. Μπορούμε να ρυθμίσουμε το HLS ώστε να αποθηκεύσει τις τιμές των σημάτων αυτών και να τις ελέγξουμε με κάποιο simulation tool όπως παράδειγμα ο Isim ή κάποιο 3° εργαλείο όπως το modelsim. Το HLS σου δίνει τη δυνατότητα να επιλέξεις τον simulator που θα χρησιμοποιήσει.

Αν ξεπεράσουμε και αυτό το στάδιο, θεωρητικά το σύστημά μας είναι έτοιμο. Το Vivado HLS παρέχει τη δυνατότητα διαφορετικών τύπων εξόδων του «υλικού» που παράχθηκε ανάλογα το εργαλείο που θέλουμε να ενσωματωθεί:

- IP-XACT format: Αποτελεί την κύρια επιλογή του HLS και πακετάρει το hardware μας με βάση το IP-XACT πρότυπο. Στην ουσία είναι ένα zip αρχείο που εμπεριέχει τον κώδικα σε Vhdl και Verilog (επιλογή αν θα συμπεριληφθούν και τα δυο) στο οποίο μπορούν να εισαχθούν και τα δεδομένα του developer, η ημερομηνία κλπ. Αυτό το αρχείο ενσωματώνεται στην πορεία στο Vivado μέσω του IP integrator και γίνεται ένα επιπλέον IP που μπορεί ο χρήστης να επιλέξει ανάμεσα στον κατάλογο IPs του Vivado.
- System Generator for DSP: Αφορά την εξαγωγή του IP για την χρήση είτε στο Vivado είτε στο ISE για την χρήση του στο System Generator for DSP που αποτελεί ένα εξειδικευμένο tool της Xilinx στη σουίτα Vivado (και ISE) που στοχεύει σε DSP αλγορίθμους στα FPGA της.
- Pcore for XPS: Αφορά την εξαγωγή του σε pcore για χρήση σε embedded εφαρμογές του XPS το οποίο επίσης αποτελεί μέρος της σουίτας VDS.

Αξίζει να σημειωθεί ότι το IP-XACT αποτελεί πρότυπο (standard) σε όλη τη βιομηχανία και σκοπός είναι να προσφέρει έναν standard τρόπο ρύθμισης του εκάστοτε IP (IP configuration) με σκοπό να ευνοήσει την εύκολη ενσωμάτωσή του σε όλα τα εργαλεία που είναι συμβατά και να διευκολύνει την επαναχρησιμοποίηση του IP για διαφορετικά projects με διαφορετικές ανάγκες ως προς τις ρυθμίσεις του (IP reuse).

Συνοπτικά η διαδικασίες που περιγράψαμε συνοψίζονται στο ακόλουθο σχήμα:

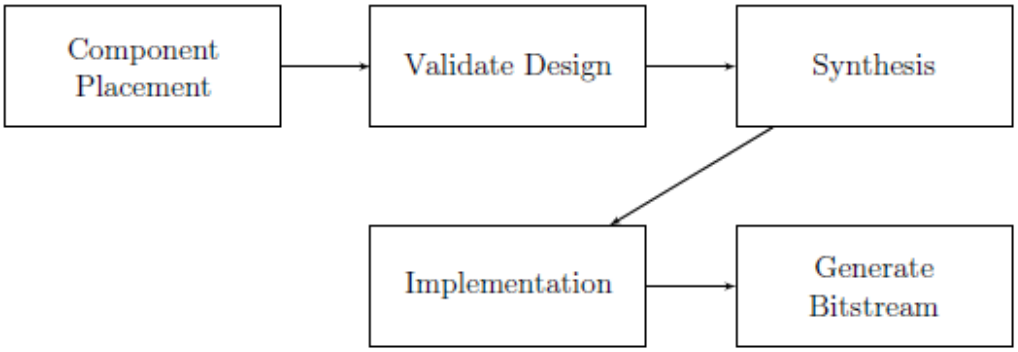


Εικόνα 48: Vivado HLS design flow

3.5.2 Vivado

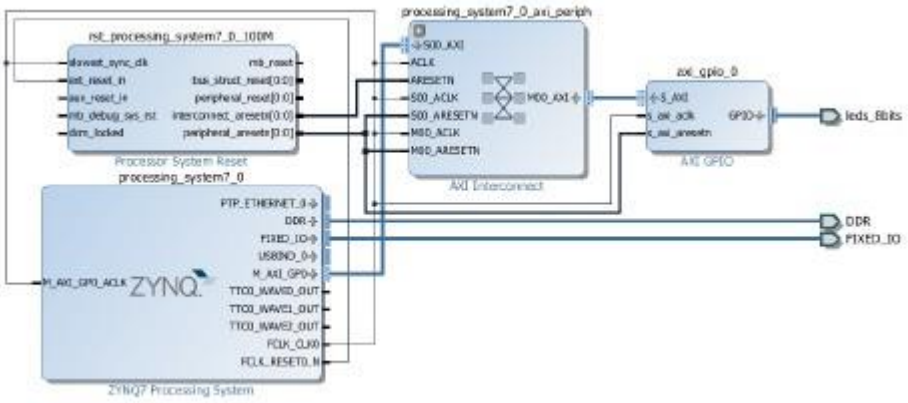
Στο περιβάλλον του Vivado γίνεται η προσομοίωση, σύνθεση και υλοποίηση του τελικού συστήματος. Παρέχει στο χρήστη πολλές επιλογές για τη σχεδίαση που επιθυμεί, μέσα από μια γραφική διεπαφή (Graphical User Interface - GUI). Εναλλακτικά, ο χρήστης μπορεί να χρησιμοποιήσει για τον ίδιο σκοπό τη γλώσσα TCL (Tool Command Language), είτε εισάγοντας εντολές στη γραμμή εντολών είτε με την εκτέλεση αρχείων TCL.

Συνοπτικά, η διαδικασία υλοποίησης ενός συστήματος στο Vivado έχει ως εξής (Σχ. 2.13):



Εικόνα 49: Vivado Design Flow

Αρχικά γίνεται η σχεδίαση του συστήματος. Μπορούμε να εισάγουμε αρχεία σε γλώσσα περιγραφής υλικού (VHDL ή Verilog), τα οποία θα ορίζουν τα δομικά στοιχεία του συστήματος μας. Εναλλακτικά μπορούμε να σχεδιάσουμε το σύστημα σε μορφή μπλοκ διαγράμματος, χρησιμοποιώντας έτοιμα IP Blocks από τις βιβλιοθήκες της Xilinx ή εισάγοντας τα δικά μας IP Blocks τα οποία έχουμε δημιουργήσει με χρήση άλλων εργαλείων του Vivado Design Suite, όπως π.χ. το Vivado HLS. Ένα παράδειγμα μπλοκ διαγράμματος δίνεται στο ακόλουθο σχήμα:



Εικόνα 50: Vivado Block Diagram

Παράλληλα, πριν τη σχεδίαση του συστήματος πρέπει να εισαχθούν κάποιοι περιορισμοί (constraints) οι οποίοι έχουν να κάνουν με ζητήματα χρονισμού (timing) και με τους ακροδέκτες του ολοκληρωμένου κυκλώματος.

Στο επόμενο βήμα γίνεται έλεγχος ορθότητας του σχεδιασμένου συστήματος (design validation), κατά τον οποίο εντοπίζονται τυχόν σφάλματα στη σχεδίαση (π.χ. κάποια λάθος σύνδεση μεταξύ στοιχείων).

Μετά τη διόρθωση των όποιων σφαλμάτων ακολουθεί το στάδιο της σύνθεσης (synthesis). Σε αυτό το στάδιο, τα αρχεία VHDL ή Verilog που περιγράφουν το σύστημα μετατρέπονται σε λίστες συνδέσεων πυλών (gate-level netlist). Αν η διαδικασία ολοκληρωθεί χωρίς σφάλματα, τότε το σύστημα είναι πλέον διαθέσιμο σε μορφή λογικών πυλών και μπορεί να προχωρήσει στο στάδιο της υλοποίησης.

Κατά την υλοποίηση (implementation), το σύστημα αρχίζει και παίρνει την επιθυμητή του μορφή. Οι λογικές πύλες που προέκυψαν από τη σύνθεση συνδυάζονται και υλοποιούνται σε στοιχεία του FPGA fabric της συσκευής, δηλ. σε LUT, FF, DSP48E1, BRAM κλπ. Οι συνδέσεις υλοποιούνται με χρήση IOBs και switch matrix.

Μετά το τέλος της υλοποίησης, μπορεί να παραχθεί το bitstream που περιγράφει το σύστημα. Το αρχείο αυτό μπορεί πλέον να φορτωθεί στο FPGA είτε άμεσα, με χρήση του JTAG, είτε έμμεσα, με την ενδιάμεση χρήση μιας εξωτερικής μνήμης flash.

3.5.3 Xilinx Software Development Kit (SDK)

Το Xilinx SDK είναι ένα ολοκληρωμένο περιβάλλον ανάπτυξης λογισμικού για ενσωματωμένα συστήματα. Είναι βασισμένο πάνω στο περιβάλλον του Eclipse IDE, το οποίο παρέχει στο χρήστη μια ευρέως γνωστή και εύληπτη γραφική διεπαφή. Σε αντίθεση με το συνηθισμένο Eclipse, το SDK παρέχει υψηλή εξειδίκευση στην ανάπτυξη ενσωματωμένου λογισμικού για ενσωματωμένα συστήματα της Xilinx, όπως π.χ. το Zynq.

Οι σημαντικότερες λειτουργίες που παρέχει το SDK είναι:

- Περιβάλλον για σύνταξη και μεταγλώττιση κώδικα C/C++ με διαχείριση σφαλμάτων

- Διαχείριση και αρχειοθέτηση projects
- Αποσφαλμάτωση πηγαίου κώδικα και ανάλυση του ενσωματωμένου στόχου
- Ο Xilinx Microprocessor Debugger (XMD), ο οποίος χρησιμοποιείται για την επικοινωνία με τους ενσωματωμένους επεξεργαστές της Xilinx μέσω του προτύπου JTAG
- Προγραμματισμός Flash, ο οποίος χρησιμοποιείται για την αυτόματη φόρτωση εφαρμογών ενσωματωμένου λογισμικού από μνήμη Flash

Το SDK υποστηρίζει εφαρμογές bare-metal (απουσία λειτουργικού συστήματος) και εφαρμογές βασισμένες σε λειτουργικό σύστημα (Linux). Στο SDK περιλαμβάνονται εργαλεία GNU όπως οι μεταγλωττιστές GCC και το GDB για αποσφαλμάτωση. Άλλο ένα χαρακτηριστικό του SDK είναι το profiling. Με τον όρο αυτό περιγράφεται μια μορφή ανάλυσης προγράμματος η οποία βοηθά στη βελτιστοποίηση μιας εφαρμογής λογισμικού, μαζεύοντας πληροφορίες σχετικές με την εκτέλεση των εντολών του προγράμματος, την χρονική και χωρική πολυπλοκότητα κ.ά. Κάποια άλλα πλεονεκτήματα του SDK είναι η χρήση πρόσθετων εφαρμογών τρίτων (plug-ins), η χρήση αποκλειστικών εργαλείων της Xilinx καθώς και η παρουσία βιβλιοθηκών πηγαίου κώδικα. Το SDK παρέχει άμεση διεπαφή με το ολοκληρωμένο περιβάλλον του Vivado που αναφέρθηκε παραπάνω.

3.6 Ενσωματωμένο Linux (Embedded Linux)

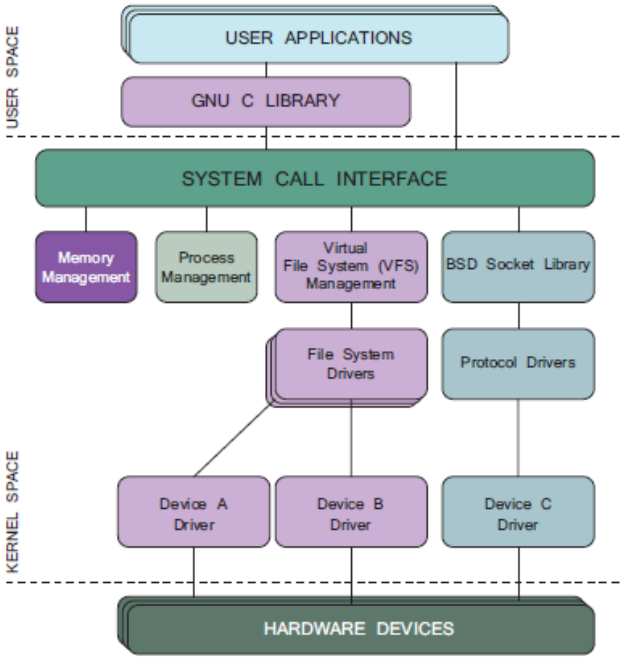
Τα ενσωματωμένα συστήματα εξελίσσονται και περιέχουν πλέον ισχυρότερους επεξεργαστές σε σχέση με το παρελθόν. Παράλληλα, συναντάται όλο και πιο συχνά η χρήση επεξεργαστή και FPGA στο ίδιο chip, όπως π.χ. στο Zynq. Συνέπεια αυτής της εξέλιξης είναι η μείωση των bare-metal εφαρμογών και η αύξηση των εφαρμογών που χρησιμοποιούν κάποιο ενσωματωμένο λειτουργικό σύστημα. Ένα από τα δημοφιλέστερα λειτουργικά για ενσωματωμένα συστήματα είναι το Embedded Linux. Το Linux είναι δωρεάν λειτουργικό ανοιχτού κώδικα (open-source) και προσφέρει μεγάλη ευελιξία στο χρήστη. Υποστηρίζει πολλές γλώσσες προγραμματισμού και βιβλιοθήκες και επιτρέπει στο χρήστη να προσαρμόσει το λειτουργικό στα μέτρα της εφαρμογής του. Επίσης, υπάρχει υποστήριξη από μια μεγάλη κοινότητα χρηστών, γεγονός που διευκολύνει την ενασχόληση με το συγκεκριμένο σύστημα. Παραδοσιακά, το ενσωματωμένο Linux είναι η προτιμότερη λύση για εφαρμογές που απαιτούν υψηλή απόδοση συστήματος. Αντίθετα, δε θεωρείται το πιο κατάλληλο για εφαρμογές που απαιτούν επεξεργασία σε πραγματικό χρόνο. Σε τέτοιες εφαρμογές προτιμάται κάποιο λειτουργικό σύστημα πραγματικού χρόνου (RTOS).

Το ενσωματωμένο Linux που χρησιμοποιείται σε εφαρμογές FPGA αποτελείται από τα εξής βασικά μέρη:

- **First Stage Bootloader (FSBL):** Στο Desktop Linux, το σύστημα εκκινεί μέσω του BIOS (Basic Input/Output System). Στα ενσωματωμένα συστήματα το ρόλο αυτό αναλαμβάνει ο FSBL. Ο FSBL είναι υπεύθυνος για λειτουργίες αρχικοποίησης, στις οποίες περιλαμβάνονται η αρχικοποίηση της κεντρικής μονάδας επεξεργασίας (CPU) και ο προγραμματισμός του FPGA με ένα bitstream. Στη συνέχεια, ο FSBL φορτώνει τον bootloader δευτέρου επιπέδου. Σε εφαρμογές χωρίς λειτουργικό σύστημα, ο FSBL θα φορτώσει κατευθείαν το πρόγραμμα του χρήστη προς εκτέλεση.
- **Second Stage Bootloader (SSBL)**: Ο SSBL αναλαμβάνει τη φόρτωση και την εκκίνηση του λειτουργικού συστήματος. Οι μικροεπεξεργαστές γενικά εκτελούν κώδικα που βρίσκεται στην τοπική μνήμη, κάτι που είναι βολικό σε εφαρμογές χωρίς λειτουργικό σύστημα, αλλά όχι σε μεγαλύτερα λειτουργικά συστήματα όπως το Linux. Το λειτουργικό σύστημα συνήθως αποθηκεύεται σε στατικές μονάδες αποθήκευσης. Επομένως, κατά την εκκίνηση του συστήματος, το

λειτουργικό σύστημα δε βρίσκεται στη μνήμη RAM. Ο SSBL είναι αυτός που αναλαμβάνει να το φορτώσει. Ο πιο δημοφιλής SSBL για συστήματα Linux είναι ο U-Boot. Μέσω του U- Boot ο χρήστης έχει τη δυνατότητα κάποιων ενεργειών πριν την εκκίνηση του λειτουργικού συστήματος, όπως η ανταλλαγή δεδομένων μέσω πρωτοκόλλου TFTP. Ο U-Boot φορτώνει τον Kernel, το Device Tree και το Root Filesystem του Linux στη μνήμη και αρχικοποιεί την εκτέλεση του Linux Kernel.

- **Kernel (Πυρήνας)**: Ο πυρήνας είναι το βασικότερο στοιχείο του λειτουργικού συστήματος. Περιλαμβάνει οδηγούς (drivers) για το υλικό του συστήματος, διαχείριση της μνήμης, διαχείριση διεργασιών, πρωτόκολλα δικτύου κ.ά. (Σχ. 2.16). Ο πυρήνας αρχικοποιεί όλο το σύστημα, καθώς και τα υποστηριζόμενα περιφερειακά, για τα οποία λαμβάνει πληροφορίες από το Device Tree. Στη συνέχεια, ο πυρήνας προχωρά στο Root Filesystem.



Εικόνα 51: Βασικά στοιχεία πυρήνα Linux

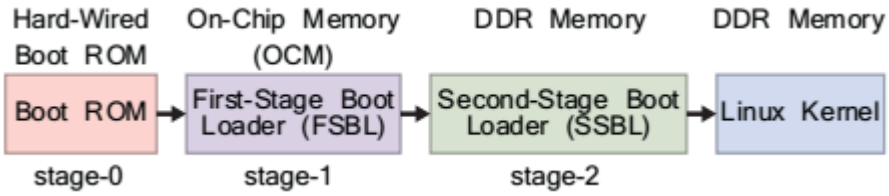
- **Device Tree:** Το Device Tree είναι ένα αρχείο που περιλαμβάνει όλο το υλικό του συστήματος που υποστηρίζεται από τον πυρήνα. Στο Desktop Linux το αρχείο αυτό είναι τμήμα του πυρήνα. Στο Embedded Linux, είναι ξεχωριστό κομμάτι, γεγονός που επιτρέπει τη φορητότητα του ίδιου πυρήνα σε συσκευές με διαφορετικό υλικό.
- **Root Filesystem:** Το σύστημα αρχείων είναι το σημαντικότερο τμήμα του λειτουργικού συστήματος μετά τον πυρήνα. Καθορίζει όλες τις λειτουργίες που μπορεί να πραγματοποιήσει το σύστημα. Παρέχει στο χρήστη τα αρχεία, τις βιβλιοθήκες και τις λειτουργίες που χρειάζεται για να εκτελέσει την εφαρμογή του.

Στον ακόλουθο πίνακα, βλέπουμε τα στάδια εκκίνησης του Linux στο Zynq

Πίνακας 6: Στάδια εκκίνησης Linux

Stage	Description
Stage-0	On power-on reset, system reset or software reset, a hard-coded boot ROM is executed on the primary processor.
Stage-1	Typically this is the FSBL. It can, however, be any user-controlled code.
Stage-2	Typically this is the user design that will run on the processing system. It could also be the second-stage bootloader, and is completely within user control.

Στη συνέχεια τα αντιστοιχούμε στο hardware του Zynq όπου αυτά υλοποιούνται:

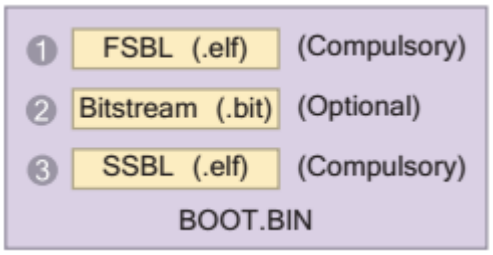


Εικόνα 52: Αντιστοίχιση Linux Boot Stages - Hardware

Για να εκκινήσει το Zynq χρειάζεται τα εξής 4 αρχεία στη μνήμη εκκίνησης:

1. BOOT.BIN
2. zImage
3. devicetree.dtb
4. ramdisk8M.image.gz

Το BOOT.BIN αρχέιο αποτελεί το boot image αρχείο. Είναι ο συνδιασμός 3 αρχείων Των fsbl και ssbl αρχείων που περιγράψαμε προηγουμένως, καθώς και του bitstream αρχείου που περιέχει το hardware της PL. Το bitstream **ΔΕΝ** είναι υποχρεωτικό καθώς το FPGA μπορεί να τρέξει και χωρίς PL. Τα υπόλοιπα είναι.



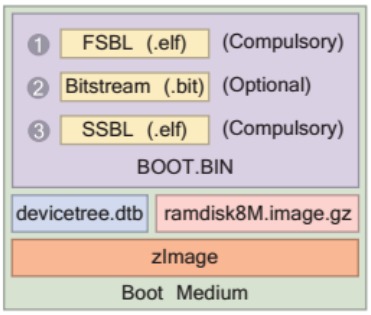
Εικόνα 53: Επιμέρους στοιχεία BOOT.BIN

Το αρχείο ***zImage*** περιέχει τον πυρήνα του Linux συμπιεσμένο. Θα αποσυμπιεστεί αυτόματα όταν φορτωθεί στη μνήμη από τον SSBL.

Το αρχείο devicetree.dtb περιέχει πληροφορίες προς τον U-Boot σχετικά με το είδος της συσκευής που θα εκτελεστεί.

Το τελευταίο αρχείο, ***ramdisk8M.image.gz***, ισόταν φορτωθεί, επιτρέπει ένα μέρος της μνήμης να αναγνωρίζεται από το Linux ως δίσκος ώστε το Linux να μπορεί να κάνει mount το root directory.

Στην εικόνα βλέπουμε συγκεντρωτικά όλα τα απαραίτητα αρχεία για την εκκίνηση του Zynq

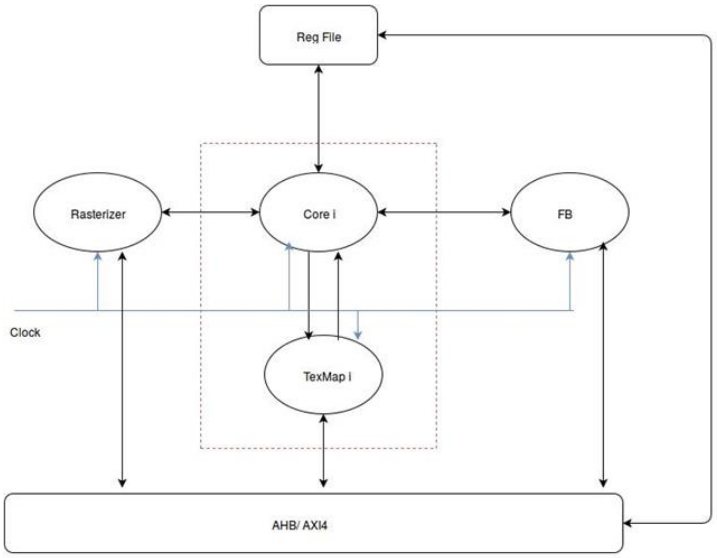


Εικόνα 54: Συγκεντρωτικά αρχεία για εκκίνηση Zynq

4. ΥΛΟΠΟΙΗΣΗ

4.1 Αρχιτεκτονική NemaP GPU

Το Nema είναι ένα σύστημα επεξεργασίας γραφικών (Graphics Processing Unit, GPU) που κατασκευάστηκε από την εταιρεία Think Silicon, Ο NemaP αποτελείται από τον Rasterizer, το σύνολο των καταχωρητών (Register File), την μνήμη πλαισίων (Frame Buffer, FB), τους επεξεργαστικούς πυρήνες (cores) και την μονάδα απεικόνισης υφών (Texture Mapping, TexMap) και το AHB/AXI σύστημα επικοινωνίας εντός του ολοκληρωμένου [14]. Στο σύστημα που υλοποιήσαμε το core έχει την δική του TexMap. Πιο συγκεκριμένα, ο Rasterizer παίρνει σαν είσοδο τις συντεταγμένες (x, y) στο επίπεδο του εικονιζόμενου στοιχείου και παράγει τα fragment pixels που το απεικονίζουν στην οθόνη. Στην περίπτωση που υπάρχει texture στην εικόνα που πρέπει να απεικονιστεί, ο rasterizer αναλαμβάνει να μετασχηματίσει σωστά τις συντεταγμένες του στην οθόνη (screen coordinate) ώστε να απεικονιστεί σωστά το εν λόγω texture (μέσω της matmult). Για κάθε fragment που παράγει ο Rasterizer σηκώνεται ένα νήμα (thread) στον core, το οποίο εκτελεί τον fragment shader, σε αυτό το βήμα τα fragments παίρνουν το τελικό τους χρώμα, fragment/pixel shading. Ο υπολογισμός της τελικής τιμής του χρώματος μπορεί να ποικίλλει από τη χρήση της interpolated τιμής χρώματος μέχρι τον υπολογισμό πολύπλοκων εξισώσεων, οι οποίες ενσωματώνουν την περιγραφή γεωμετρίας και εμφάνισης του αντικειμένου και την περιγραφή του περιβάλλοντος. Αυτοί οι υπολογισμοί μπορεί να περιλαμβάνουν λειτουργίες αντιστοίχισης υφής από την αντίστοιχη TexMap του core. Οι λειτουργίες αυτές χρησιμοποιούν τις παραμετρικές συντεταγμένες του pixel fragment για να κάνουν δειγματοληψία σε μια κατάλληλη εικόνα, η οποία ονομάζεται texture, ώστε να δημιουργήσει μια τιμή για το χρώμα ή άλλες παραμέτρους. Τα σκιασμένα πλέον pixel fragments γράφονται τελικά στη μνήμη χρωμάτων, η οποία κρατάει μόνο ένα fragment για κάθε pixel στην τελική εικόνα. Σαν μέρος αυτής της λειτουργίας, μπορεί να γίνουν κάποιες άλλες λειτουργίες σχετικές με τα γραφικά (π.χ. να αποφασιστεί η ορατότητα του pixel χρησιμοποιώντας έναν ειδικό χώρο αποθήκευσης, τη μνήμη βάθους). Στην μνήμη πλαισίων, FB, αποθηκεύονται οι προσθήκες χρώματος που προκύπτουν τις λειτουργίες αυτές. Η αρχιτεκτονική του συστήματος φαίνεται αναλυτικά στο παρακάτω σχήμα.



Εικόνα 55: Αρχιτεκτονική Nema GPU

4.2 Περιγραφή αλγορίθμων που υλοποιήθηκαν

Εφόσον αναφέραμε τους βασικούς αλγόριθμους που χρησιμοποιούνται σήμερα στο rasterization μπορούμε να αναφέρουμε τι υλοποιήθηκε κατά τη διάρκεια της διπλωματικής. Σκοπός ήταν να υλοποιηθεί ο Rasterizer της Nema GPU. Η εταιρεία διαθέτει διαφορετικές εκδόσεις της κάρτας, με διαφορετικές δυνατότητες ενσωματωμένες που εξυπηρετούν διαφορετικές ανάγκες των πελατών της για διαφορετικές συσκευές. Για το verification της HLS έκδοσής του rasterizer χρησιμοποιήθηκε η «μικρή» έκδοσή της που ονομάζεται NemaP.

Οι αλγόριθμοι rasterization που χρησιμοποιούμε είναι μία βελτιστοποιημένη έκδοση του αλγορίθμου edge function ελαφρά διαφορετικό από τον τρόπο υπολογισμού του αλγορίθμου που περιγράψαμε και αρκετά όμοιο με τον τρόπο που ο Pidela προτείνει, όπου χρησιμοποιούμε μόνο στην αρχή τον υπολογισμό των edge functions, και στην συνέχεια «περπατάμε» στο τρίγωνο με προσθέσεις και αφαιρέσεις. Ο ίδιος αλγόριθμος χρησιμοποιείται και στα χρώματα. Ο rasterizer λαμβάνει ως είσοδο το χρώμα μόνο ενός σημείου (PT0) καθώς και τιμές για τα αντίστοιχα dx και dy. Οι τιμές αυτές προκύπτουν από τις επιμέρους τιμές των σημείων που ορίζουν το σχήμα προς σχεδίαση και υπολογίζονται στον μικροεπεξεργαστή που συνοδεύει την κάρτα γραφικών. Στις μεγαλύτερες εκδόσεις του Nema οι πράξεις αυτές γίνονται εντός της GPU στον Vertex processor (core). Το κέρδος από αυτήν την επιλογή Interpolation αλγορίθμων είναι η πολύ μικρή πολυπλοκότητα και κατανάλωση ενέργειας που προκύπτει από αυτό.

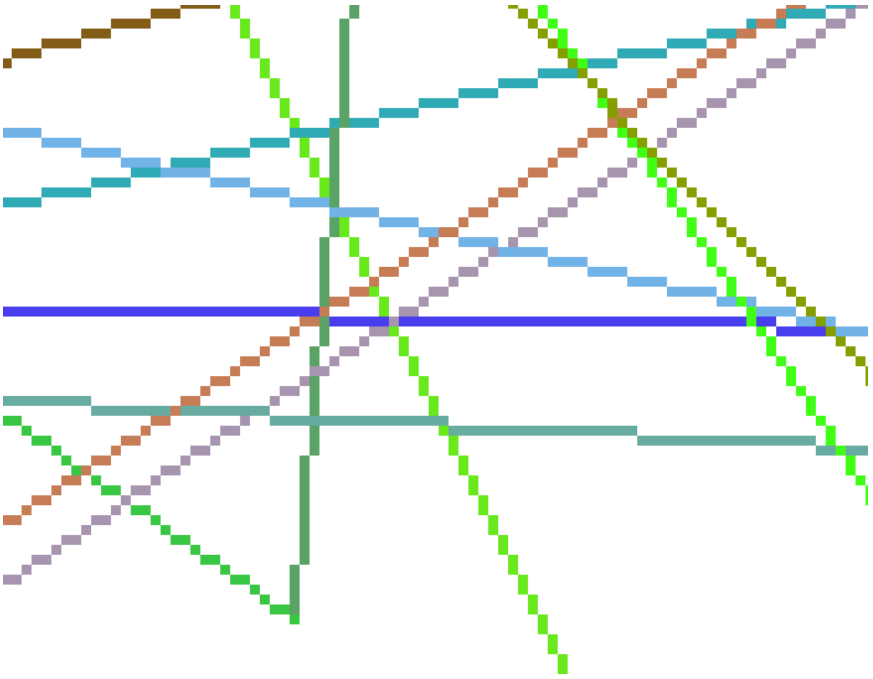
Στην περίπτωση βέβαια του texturing ακολουθούμε τον perspective correct interpolation για να έχουμε αξιόπιστα αποτελέσματα. Στην περίπτωση των γραμμών, ακολουθείται ο αλγόριθμος του Bresenham.

Στη συνέχεια παραθέτουμε μερικές εικόνες από τα tests που έγιναν για την εξακρίβωση της ορθής λειτουργίας του rasterizer:

4.3 Εξακρίβωση ορθής λειτουργίας του Rasterizer

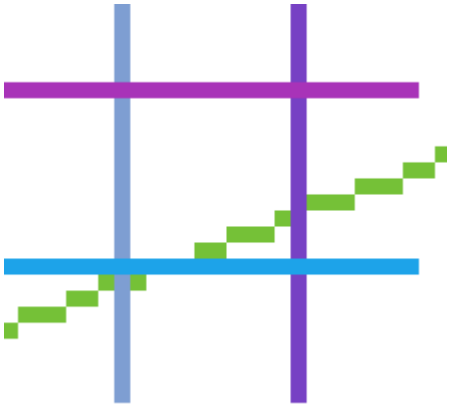
4.3.1 Line Drawing Bresenham

Στην παρακάτω εικόνα, είναι μεγεθυμένη σε τέτοιο βαθμό ώστε να είναι εμφανής ο τρόπος «λειτουργίας» του αλγορίθμου του Bresenham.



Εικόνα 56: Πλάγιες γραμμές σχεδιασμένες με τον Bresenham algorithm

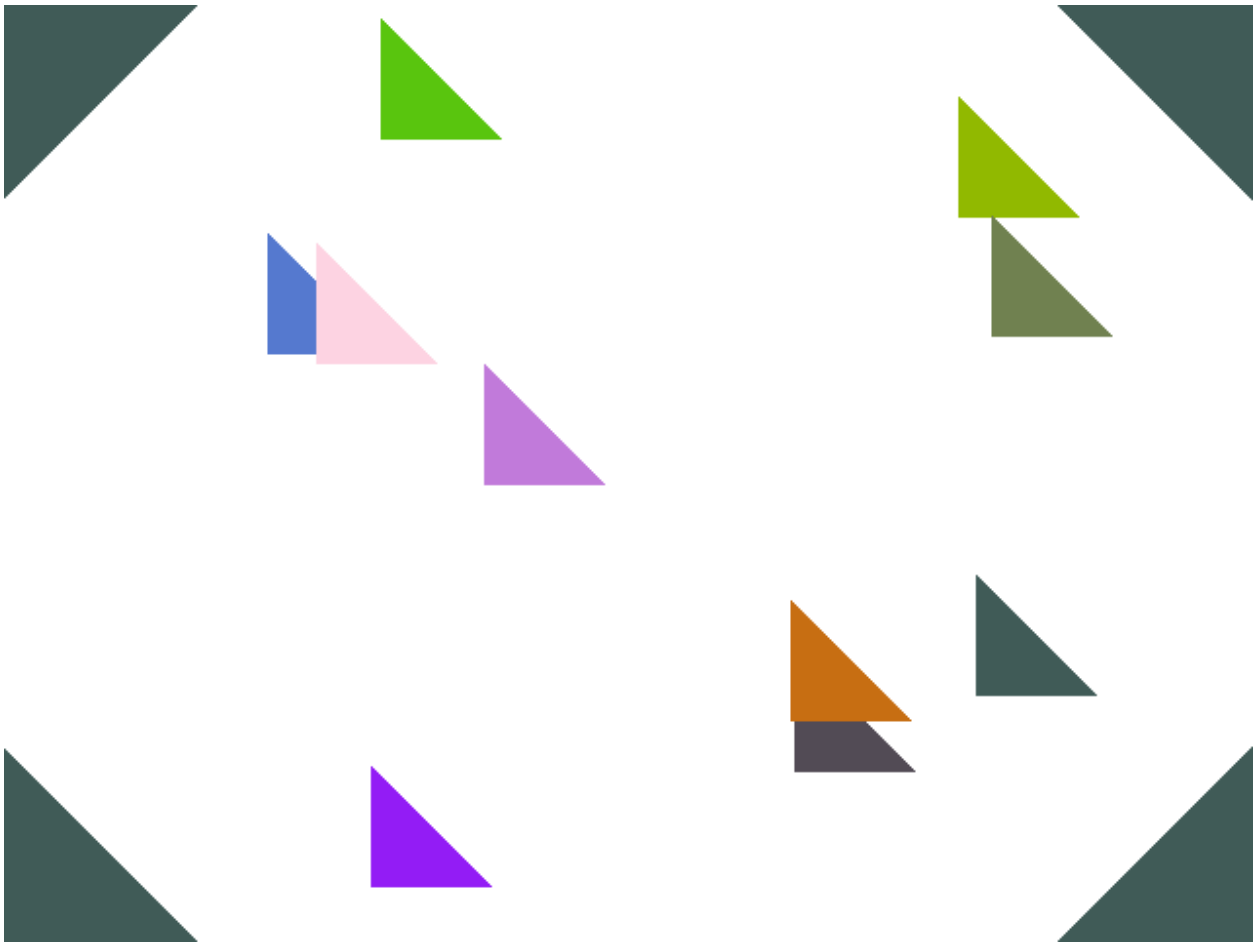
Στην ακόλουθη εικόνα, έχουμε εστιάσει στις κάθετες και οριζόντιες γραμμές. Σύμφωνα με τον αλγόριθμο αυτές οι γραμμές δεν παρήγαγαν λάθος (error) οπότε και το αποτέλεσμα σε αυτές είναι σαφώς καλύτερο οπτικά



Εικόνα 57: Οριζόντιες και κάθετες γραμμές σχεδιασμένες με τον Bresenham algorithm

4.3.2 Triangle Drawing

Στην συνέχεια βλέπουμε την 1η εικόνα που σχεδίασε σωστά ο rasterizer και αφορά το σημαντικότερο σχήμα που υλοποιεί, το τρίγωνο.



Εικόνα 58: Ολοκληρωμένη εικόνα σχεδιασμού τριγώνων

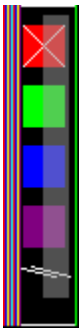
Αρχικά σχεδιάσαμε 10 τρίγωνα, τα οποία είχαν όλα ενιαίο αλλά διαφορετικό χρώμα. Σκοπός μας ήταν να εξακριβώσουμε ότι λειτουργεί σωστά η Edge function και ότι σχηματίζει χρωματίζει σωστά τα pixels.

Στη συνέχεια μεγεθύνουμε ένα τυχαίο τρίγωνο της προηγούμενης εικόνας ώστε να παρατηρήσουμε καλύτερα τον τρόπο που το τρίγωνο σχεδιάζεται κατά γραμμή. Όπως παρατηρούμε και πάλι στις κάθετες και οριζόντιες πλευρές δεν έχουμε pixelάρισμα ενώ στην υποτίνουσα το φαινόμενο αυτό είναι εμφανές. Παρόλα αυτά οι θέσεις και τα χρώματα των pixels είναι τα αναμενόμενα

Εικόνα 59: Μεγένθυνση τριγώνου για εξακρίβωση λειτουργίας αλγορίθμου

4.3.3 Texturing

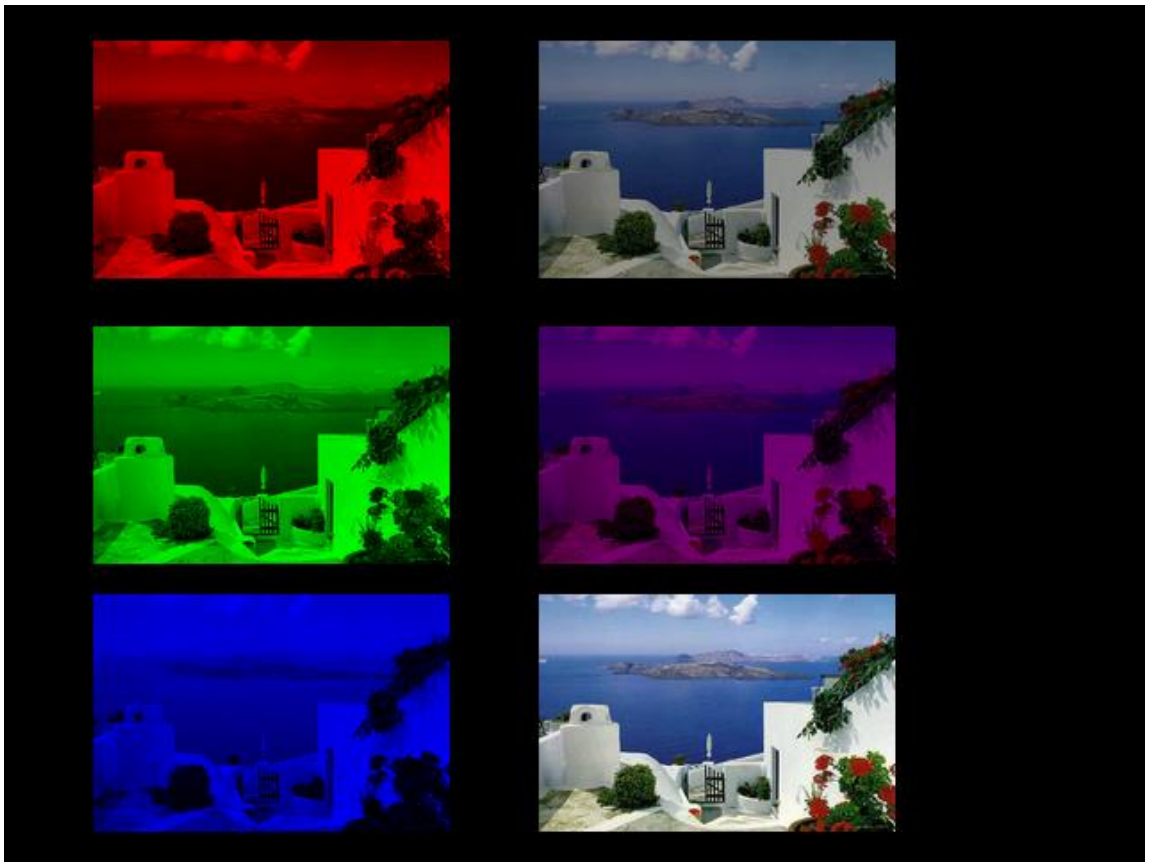
Εδώ εξετάζουμε κυρίως, τη λειτουργία του δεύτερου κομματιού της υλοποίησής μας, που ονομάζεται matmult. Το κομμάτι αυτό του IP εκτελεί τις πράξεις που απαιτούνται για τον υπολογισμό των texture συντεταγμένων (texture coordinates).



Εικόνα 60: Συνδιασμός Γραμμών, rectangles και texturing

Το συγκεκριμένο σχήμα αν και εκ πρώτης όψεως είναι απλό, στην ουσία εξετάζει τα περισσότερα κομμάτια του rasterizer. Αριστερά και δεξιά σχεδιάζει κάθετες γραμμές, στο κέντρο σχεδιάζει rectangles (ορθογώνια που ορίζονται από δυο σημεία και με βάση αυτά υπολογίζονται τα υπόλοιπα) και κυρίως ελέγχεται η λειτουργία της matmult εφαρμόζωντας το texture που βλέπουμε στα δεξιά της εικόνας. Το texture αυτό αφορά κυρίως τη διαφάνεια.

Στην επόμενη εικόνα βλέπουμε την εφαρμογή διαφορετικών textures σε μια εικόνα της Σαντορίνης.



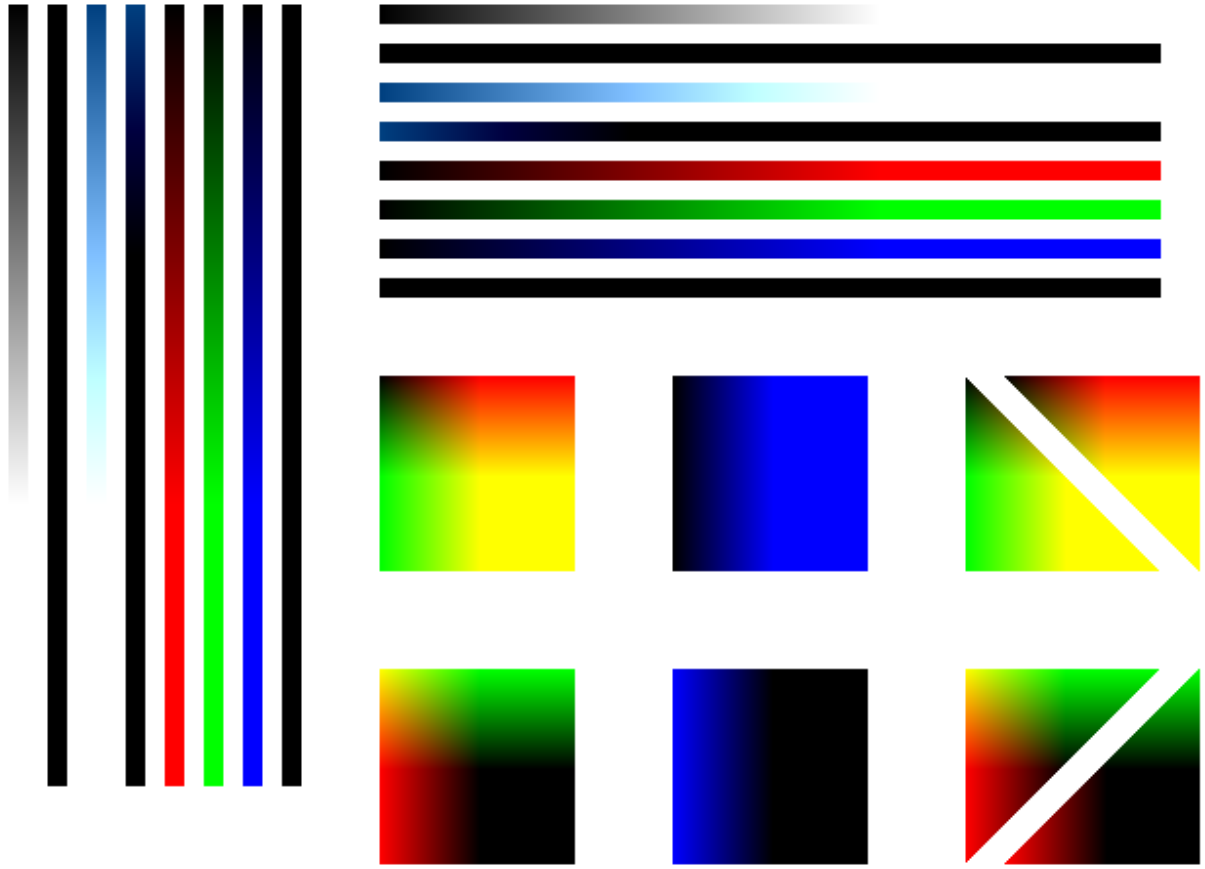
Εικόνα 61: Χρωματισμός της εικόνας της Σαντορίνης με τη χρήση texturing

Ομοίως και στην επόμενη εικόνα βλέπουμε την εφαρμογή διαφάνειας ως texture που έχει ως αποτέλεσμα την «εξαφάνιση» της εικόνας όταν φτάσει στην μέγιστη τιμή του

Εικόνα 62: Εφαρμογή texture διαφανοποίησης στην εικόνα της Σαντορίνης

4.3.4 Mixed Shapes

Και τέλος μια εικόνα στην οποία εμφανίζονται όλα τα σχήματα που σχεδιάζει ο rasterizer καθώς και το interpolation που υλοποιείται:



Εικόνα 63: Συνδυασμός όλων των σχημάτων και χρωματισμός με γραμμικό αλγόριθμο

4.3.5 Σύγκριση Αποτελεσμάτων

Η εργασία εκπονήθηκε με σκοπό τη σύγκριση των αυτόματων λύσεων που προσφέρει ένα λογισμικό τύπου HLS και των custom υλοποιήσεων που μπορεί να επιτύχει ένας μηχανικός. υλοποιώντας ακριβώς τους ίδιους αλγορίθμους σε επίπεδο RTL. Σε αυτήν την ενότητα συγκρίνουμε τις υλοποιήσεις:

Τα παρακάτω αποτελέσματα που αφορούν τον υπάρχοντα rasterizer της εταιρίας για το zynq.

Πίνακας 7: Πόροι Rasterizer σε Verilog

Primitive Statistics

Primitive type	Count
FLOP_LATCH	3721
LUT	8999
MUXFX	175
CARRY	712
MULT	6
DMEM	217
OTHERS	351

Τα παρακάτω στοιχεία αφορούν τη δική μας υλοποίηση του rasterizer με τη χρήση του Vivado HLS:

Πίνακας 8: Πόροι Rasterizer σε HLS

Name	LUT	FF	BRAM	URAM	DSP
	4	3	0	0	0
tsi_rasterize_0_synth_1	13124	5880	0	0	84
mat_mult_only_0_synth_1	14659	13011	0	0	90
fifo_generator_0_synth_1	374	379	0	0	0

Συγκεντρώνοντάς τα σε έναν πίνακα για πιο εύκολη σύγκριση.

Πίνακας 9: Συγκεντρωτικός πίνακας πόρων Rasterizer

Όνομα	LUT	FF	DSP/MUL	BRAM
Original Rasterizer	8999 (217 Dmem)	3721	6 (712 carry)	0
Tsi_rasterize	13124	5880	84	0
mat_mult	14659	13011	90	0
FiFO	374	379	0	0
Total HLS	28157	19270	174	0

Όπως αντιλαμβανόμαστε εκ πρώτης όψεως η custom υλοποίηση καταναλώνει αρκετά λιγότερους πόρους. Η μεγαλύτερη διαφορά δείχνει να είναι στους πολλαπλασιαστές/DSP αφού ο original χρησιμοποιεί μόλις 6 ενώ η υλοποίηση του HLS 174 DSPs. Και στην περίπτωση των LUTs παρατηρούμε αύξηση άνω του 200% και στων FF άνω του 400%. Προφανώς μπορεί να γίνει βελτιστοποίηση ανάλογα με τις ανάγκες που έχουμε και χρησιμοποιώντας τις διάφορες ντιρεκτίβες που μας δίνει τη δυνατότητα το HLS αλλά εκ

πρώτης όψεως η custom λύση είναι αρκετά καλύτερη (τουλάχιστον ως πρό το area/power). Αξίζει να σημειωθεί ότι το HLS σχεδιάζει hardware με σκοπό να χρησιμοποιηθεί στα FPGA της οπότε και βελτιστοποιείται για την πλακέτα που προορίζεται και με βάση το hardware που αυτή έχει. Δηλαδή σε δυο διαφορετικές πλακέτες μπορεί να πάρουμε δυο πολύ διαφορετικά αποτελέσματα. Η περιγραφή σε Verilog/VHDL συνήθως υπονοεί με πιο μεγάλη ακρίβεια το hardware που θα χρησιμοποιηθεί. Αν ο σκοπός είναι να τρέξει σε κάποιο FPGA είναι καλό να χρησιμοποιεί τα IPs που παρέχει η Xilinx για να πάρει τη βέλτιστη απόδοση. Αν όμως επιθυμούμε ο κώδικας μας να έχει μεταφερσιμότητα (portability) και σε άλλες πλατφόρμες χωρίς ιδιαίτερες αλλαγές, τότε επιβάλλεται να χρησιμοποιούμε μόνο κώδικα ο οποίος είναι κοινός ανάμεσα στις εταιρίες δηλαδή κοινά IPs. Ο RTL κώδικας που παράγει το HLS είναι κρυπτογραφημένος σε διάφορα σημεία, άρα η οποιαδήποτε προσπάθεια χρησιμοποίησης του συνόλου του για χρήση εκτός των FPGAs της Xilinx είναι σχεδόν αδύνατη. Πάντως ακόμα και τα κομμάτια του κώδικα που δεν είναι κρυπτογραφημένα, εξαιρώντας κάθε φορά το interface των εξωτερικών blocks είναι πολύ δυσανάγνωστα και είναι σχεδόν αδύνατον κάποιος να καταλάβει τις διαδικασίες που γίνονται εκτός και αν αφορά κάτι πολύ απλό. Το HLS είναι ένα πολύ χρήσιμο εργαλείο για την ανάπτυξη υλικού σε σχετικά μικρό χρονικό διάστημα, ειδικά στην περίπτωση που αποκτήσεις κάποια οικειότητα με το εργαλείο. Οι περιορισμοί όμως που έχει ειδικά στην περίπτωση της μεταφερσιμότητας αλλά και τις συνεργασίας με παλιότερα εργαλεία μπορούν να αποτελέσουν ανασταλτικούς παράγοντες για τη χρήση του.

Κατά την εκπόνηση παρουσιάστηκε πλήθος δυσκολιών ενσωμάτωσης του παραχθέντος υλικού με τα υπόλοιπα εργαλεία της εταιρίας και αυτό απαίτησε ένα σημαντικό μέρος του χρόνου εργασίας.

Το ίδιο το HLS είναι ένα εργαλείο που συνεχώς εξελίσσεται και σε κάθε νέα έκδοση προστίθενται νέα χαρακτηριστικά (πχ τύπος half (float) στις μεταβλητές) καθώς και διορθώνονται διάφορα bugs προγούμενων εκδόσεων. Αν και θεωρητικά κάθε IP είναι συμβατό με επόμενες εκδόσεις του λογισμικού, ειδικά αν ακολουθεί το πρότυπο IP-XACT, συνήθως είναι καλύτερα να ακολουθείται η αντιστοιχία έκδοσης του Vivado με αυτή του HLS και στην περίπτωση 3^{ων} εργαλείων για προσομοίωση, σύνθεση κλπ πρέπει να εξετάζουμε τις συμβατές εκδόσεις που ορίζει η Xilinx με σκοπό την αποφυγή όσο των δυνατών περισσότερων bugs. Στη δική μας περίπτωση χρησιμοποιήθηκαν οι εκδόσεις 2015.1 σε συνεργασία με το Qustasim 10.4.

Η λογική της εταιρικής χρήσης του rasterizer οδήγησε στην προσπάθεια εξεύρεσης αυτοματοποιημένων λύσεων δημιουργίας, σύνθεσης κλπ ώστε η όλη διαδικασία να είναι κρυμμένη από τον οποιονδήποτε θέλει να χρησιμοποιήσει την HLS υλοποίηση του rasterizer για να εξετάσει οποιαδήποτε διαφορά στην απόδοση-απεικόνιση κλπ. Για αυτόν τον λόγο χρησιμοποιήθηκε ο συνδιασμός bash scripting και tcl scripting και git versioning control για την τελική δημιουργία του προς εξέταση κώδικα (είτε για simulation είτε για implementation).

4.4 Implementation

Στην περίπτωση του Vivado HLS έχουμε σχετικά μικρό έλεγχο στο πως το ίδιο το σύστημα θα υλοποιήσει το κάθε τι. Ευτυχώς η Xilinx με τα directives μας δίνει τη δυνατότητα να δίνουμε «εντολές» σχετικά με τον τρόπο υλοποίησης, καταλήγοντας να μπορούμε να ελέγξουμε καλύτερα το hardware που θα χρησιμοποιηθεί και κυρίως επιτυγχάνοντας πιο προβλέψιμα αποτελέσματα από πλευράς timing. Πιο πρακτικά:

Για το texturing χρησιμοποιήσαμε perspective correct interpolation. Στην υλοποίηση του αλγορίθμου κάναμε επί της ουσίας τις ακόλουθες πράξεις:

$$DET = t20 * (x + added_value) + t21 * (y + added_value) + t22$$

$$tx = (t00 * (x + added_value) + t01 * (y + added_value) + t02) / DET$$

$$ty = (t10 * (x + added_value) + t11 * (y + added_value) + t12) / DET$$

Όπου τα αποτελέσματά μας όπως φαίνεται είναι οι texturing coordinates tx, ty και τα δεδομένα μας είναι οι τιμές του πίνακα texturing t00-t22 τα x, y που είναι οι συντεταγμένες του pixel που έχουμε υπολογίσει στην tsi_rasterize καθώς και η added_value που είναι μια directive η οποία δεν αλλάζει κατά τη διάρκεια «εκτέλεσης» οπότε την σκεφτόμαστε συνήθως σαν μια τιμή είτε 0 είτε fixed point (πχ 0.5). Όπως μπορούμε να διακρίνουμε δεν υπάρχει καμία εξάρτηση των εξόδων (tx, ty) από κάποια loop. Όμως από την ανάλυση που παίρνουμε από το hls:

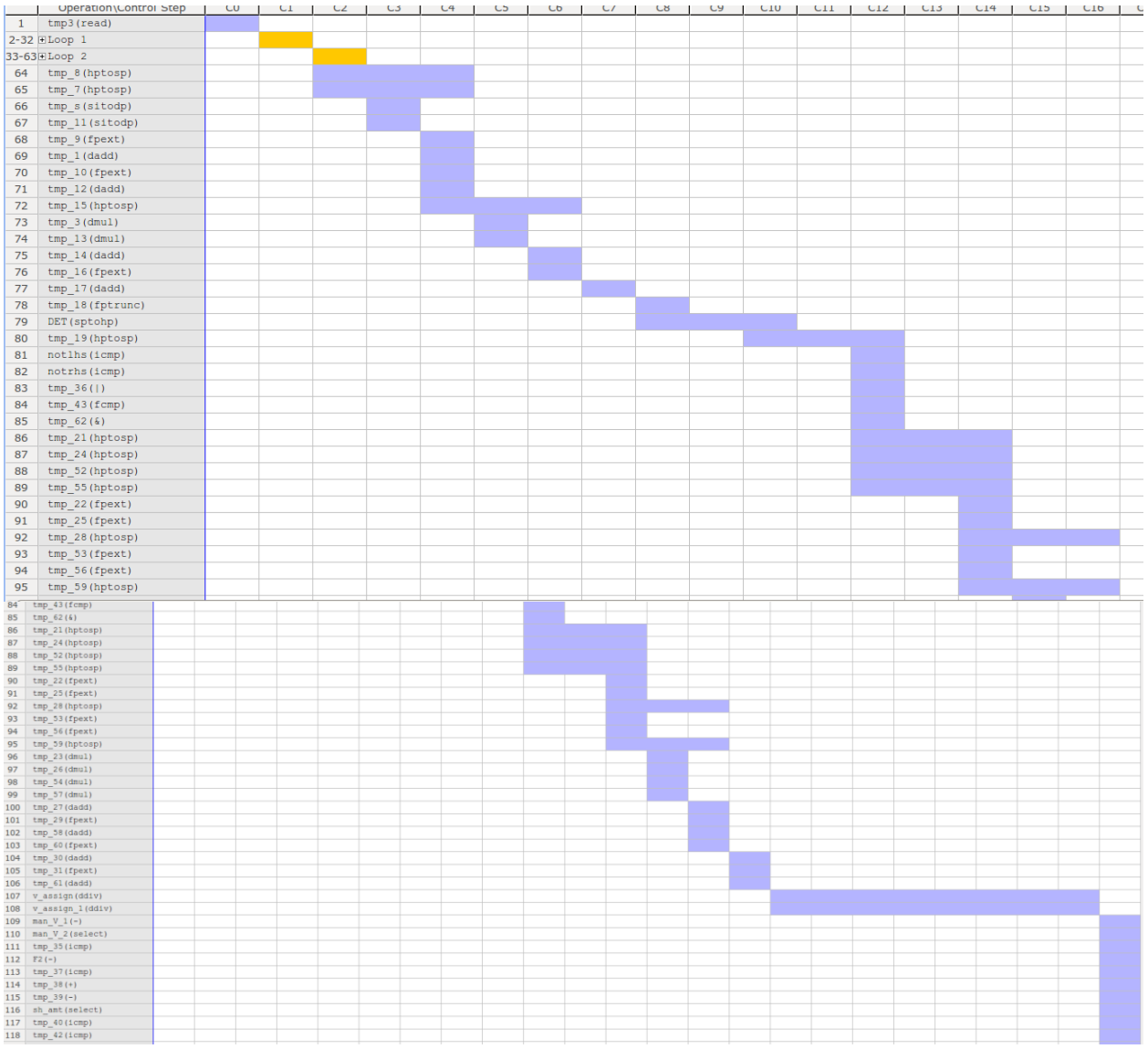
	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
mat_mult_only	0	50	6089	15869	11~34	12 ~ 35	none

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
mat_mult_only	-	11~34	12 ~ 35	-	-
Loop 1	no	4	-	1	4
Loop 2	no	4	-	1	4

Εικόνα 64: Matmult Resources without Pipelining

Το HLS σου δίνει την δυνατότητα να δεις πως χρησιμοποιεί το hardware σε κάθε κύκλο. Συγκεκριμένα για τις παραπάνω πράξεις έχουμε αυτήν την ανάλυση:



Εικόνα 65: HLS time scheduling

Όπως μπορούμε να αντιληφθούμε, δεν μπορούμε να αντιληφθούμε πάντα επακριβώς τι εκτελεί το HLS σε κάθε κύκλο. Τα ονόματα μας δίνουν μια ιδέα αλλά δεν μπορούμε να είμαστε πάντα σίγουροι. Δηλαδή, έχουμε mul, div, add κλπ αλλά και πάλι λόγω των ονομάτων σε πιο μεγάλα προγράμματα δεν ξέρουμε σε ποιο σημείο του κώδικα απευθύνονται. Πάντως στην συγκεκριμένη περίπτωση αντιλαμβανόμαστε, ότι κομμάτια του κώδικα που δεν έχουν εξαρτήσεις μεταξύ τους, δεν εκτελούνται παράλληλα. Η προφανής επιλογή θα ήταν όλοι οι πολλαπλασιασμοί να εκτελούνται στον ίδιο κύκλο. Όμως όπως παρατηρούμε δεν ισχύει κάτι τέτοιο. Και σίγουρα τα 50 DSPs που η σύνθεση μας λέει ότι θα χρειαστούμε απέχουν από μια οικονομική προσέγγιση του προβλήματος. Κάτι άλλο που αξίζει να σημειωθεί είναι ότι καταναλώνονται (αρκετοί) κύκλοι για την μετατροπή των floating point σε fixed point και το αντίστροφο καθώς τα x, y είναι σε fixed point και τα αποτελέσματα tx, ty, t00-t22 σε floating point.

Το οποίο σημαίνει ότι δεν έχουμε pipeline και συνεπώς για να υπολογίσουμε μια καινούρια τιμή tx, ty θα πρέπει να «περιμένουμε» 12-35 κύκλους εφόσον κάθε υπολογισμός διαρκεί 11-34 κύκλους. Όπως αντιλαμβανόμαστε κάτι τέτοιο θα ήταν καταστροφικό εφόσον μιλάμε για pixels δηλαδή streaming δεδομένα, όπου το pipelining

είναι απαραίτητο. Ευτυχώς έχουμε τη δυνατότητα να δώσουμε την directive pipeline II =1 στο HLS ώστε να προσπαθήσει να κάνει pipeline στο hardware.

Χρησιμοποιώντας την παραπάνω δυνατότητα, καταλήγουμε στην εξής υλοποίηση:

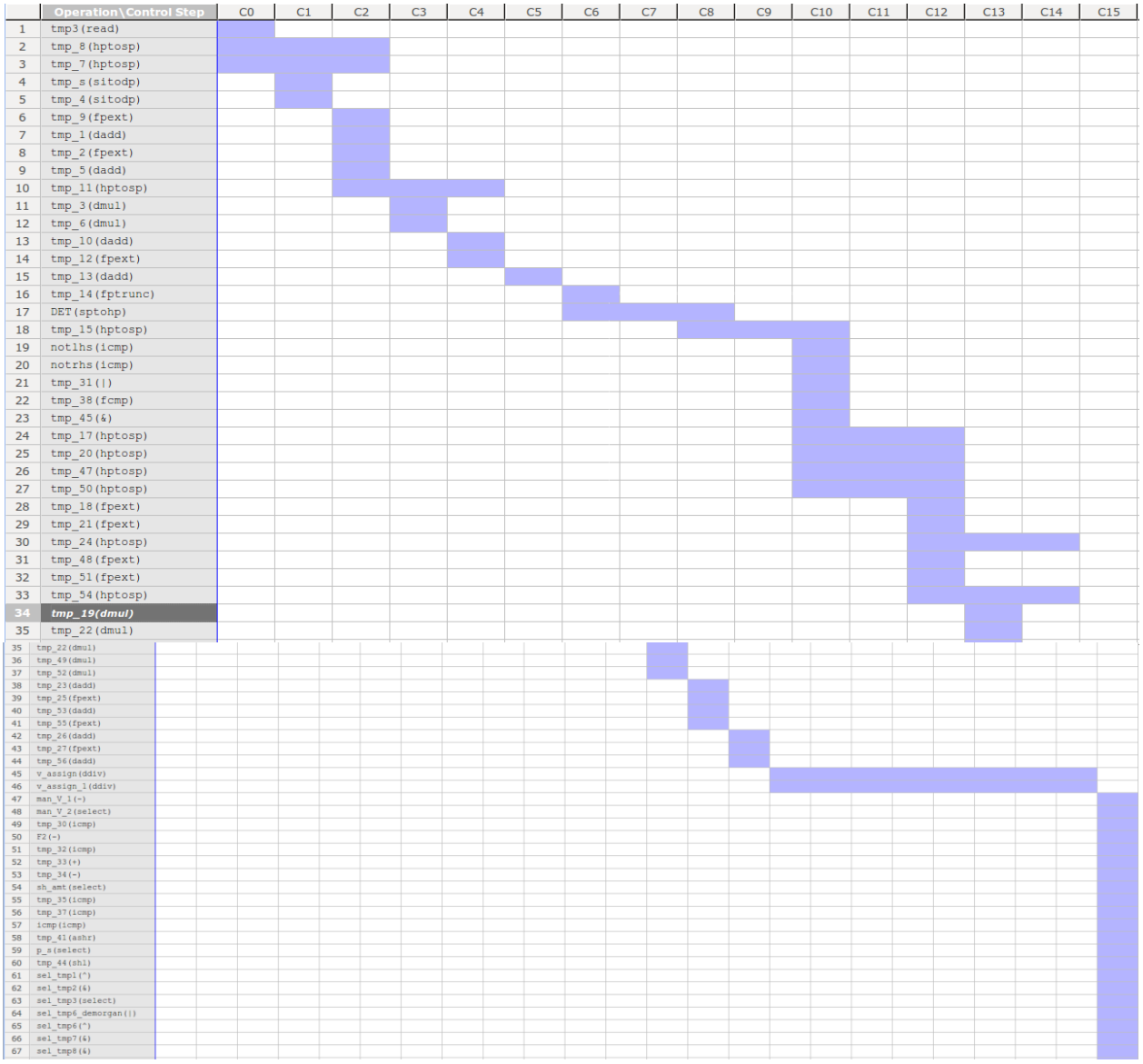
	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
mat_mult_only	0	90	21245	23230	24	1	function

Performance Profile | Resource Profile

	Pipelined	Latency	Initiation Interval	Iteration Latency	Trip count
mat_mult_only	-	24	1	25	-

Εικόνα 66: HLS Matmult Resources and timing

Όπως παρατηρούμε, πλέον έχουμε σταθερό latency 24 κύκλους ενώ το σύστημά μας είναι πλήρως pipelined. Αυτό σημαίνει ότι ανά κύκλο μπορούμε να επεξεργαζόμαστε και ένα καινούριο pixel. Έτσι και η ανάλυση που μας δίνει το hls αλλάζει στο ακόλουθο…



Εικόνα 67: HLS Pipelined time scheduling

Πλέον το κύκλωμά μας διαπιστώνουμε ότι λειτουργεί με πολύ πιο προβλέψιμο τρόπο και η ανάλυσή του είναι και ελαφρά πιο κατανοητή. Παρατηρούμε επίσης ότι οι DSP από 50 αυξήθηκαν σε 90. Πολύ μεγάλη αύξηση στην ήδη «κακή» από πλευράς resources υλοποίηση του HLS. Παρόλα αυτά βλέπουμε πως μπορούμε να «φέρουμε» την υλοποίηση πιο κοντά στις απαιτήσεις μας. Ανάλογα με το ποιες είναι αυτές, μπορούμε να χρησιμοποιήσουμε οποιαδήποτε από τις directives καθώς και ενδεδειγμένα από την Xilinx coding styles, ώστε να πετύχουμε είτε υψηλότερη συχνότητα είτε λιγότερα resources είτε λιγότερο latency. Προφανώς το γεγονός ότι εμείς θα θέσουμε κάποιο όριο, δεν σημαίνει ότι το HLS θα μπορέσει να το υλοποιήσει. Για παράδειγμα για τον συγκεκριμένο κώδικα, το HLS δεν ήταν δυνατόν να τον υλοποιήσει με max latency 8, αλλά δεν είχε πρόβλημα να υλοποιήσει τον κώδικα με περίοδο 20ns (50 MHz) αντί των 20 Mhz (50 ns) του αρχικού design. Δεν χρειάστηκε περισσότερα DSP αλλά αύξησε το latency σε 57 (από 24). Πάντως δεδομένου ότι στο συγκεκριμένο design μας ενδιαφέρει πολύ περισσότερο το pipelining και η περίοδος και όχι το latency, θα λέγαμε ότι το HLS θα μπορούσε να αποτελέσει ένα πολύ βοηθητικό εργαλείο.

Όπως εξηγήσαμε σε προηγούμενες σελίδες σχετικά με τον κώδικα που παράγει το HLS, είναι σχετικά ακατανόητος. Όπως βλέπουμε σε αποσπάσματα του παρακάτω κώδικα, αν εξαιρέσουμε το εξωτερικό interface, ο υπόλοιπος κώδικας είναι σχετικά ακατανόητος.

```
input   ap_clk;
input   ap_rst;
input   ap_start;
output   ap_done;
output   ap_idle;
output   ap_ready;
input  [355:0] pixel_in_V_dout;
input   pixel_in_V_empty_n;
output   pixel_in_V_read;
output  [355:0] pixel_out_V_din;
input   pixel_out_V_full_n;
output   pixel_out_V_write;

reg ap_done;
reg ap_idle;
reg ap_ready;
reg pixel_in_V_read;
reg pixel_out_V_write;
(* fsm_encoding = "none" *) reg   [0:0] ap_CS_fsm0 = 1'b1;
reg    ap_sig_cseq_ST_st1_fsm0_0;
reg    ap_sig_bdd_89;
(* fsm_encoding = "none" *) reg   [1:0] ap_CS_fsm1 = 2'b1;
reg    ap_sig_cseq_ST_st0_fsm1_0;
reg    ap_sig_bdd_100;
(* fsm_encoding = "none" *) reg   [1:0] ap_CS_fsm2 = 2'b1;
reg    ap_sig_cseq_ST_st0_fsm2_0;
reg    ap_sig_bdd_108;

parameter    ap_const_lv32_B4 = 32'b10110100;
parameter    ap_const_lv32_C3 = 32'b11000011;
parameter    ap_const_lv32_C4 = 32'b11000100;
parameter    ap_const_lv32_D3 = 32'b11010011;
parameter    ap_const_lv32_D4 = 32'b11010100;
parameter    ap_const_lv32_E3 = 32'b11100011;
parameter    ap_const_lv32_E4 = 32'b11100100;


mat_mult_only_fptrunc_64ns_32_1 #(
    .ID( 1 ),
```

```
    .NUM_STAGE( 1 ),
    .din0_WIDTH( 64 ),
    .dout_WIDTH( 32 ))
mat_mult_only_fptrunc_64ns_32_1_U1(
    .din0( tmp_13_reg_1130 ),
    .dout( tmp_14_fu_201_p1 )
);
```

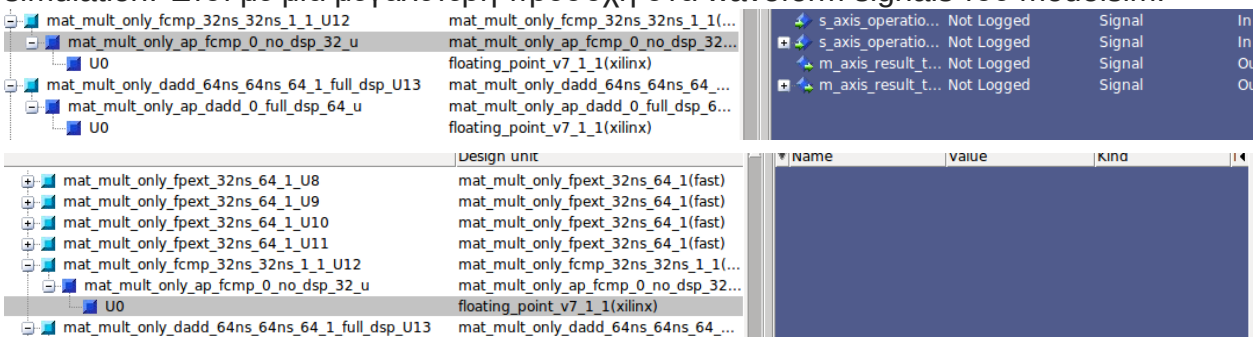
Ο συγκεκριμένος κώδικας αποτελείται από αποσπασματικά κομμάτια του κώδικα που παρήγαγε το εργαλείο για τις ανωτέρω εξισώσεις. Για να κατανοήσουμε την πολυπλοκότητα του παραγόμενου κώδικα, δεν αρκεί μόνο να παρατηρήσουμε την ονομασία, αλλά αξίζει να αναφερθεί το γεγονός ότι το κύριο αρχείο που παρήγαγε το εργαλείο ήταν ~3500 γραμμές. Υπάρχουν και άλλα ~10 αρχεία από 30-100 γραμμές το καθένα. Αλλά όπως κατανοούμε για 10-15 αριθμητικές πράξεις, η παραγωγή πλέον των 4000 γραμμών κώδικα, μας οδηγεί στο γεγονός ότι είναι αδύνατον να κάνουμε την οποιαδήποτε βελτιστοποίηση χειροκίνητα.

Ένα ακόμα χαρακτηριστικό του development με HLS είναι ότι, όπως είναι άλλωστε λογικό, γίνεται «εκτεταμένη» χρήση των βιβλιοθηκών της Xilinx γεγονός που οδηγεί από την μία σε μια efficient χρήση του FPGA από την άλλη όμως οδηγεί στο «κρύψιμο» κομματιών του κώδικα ώστε να κάνει ακόμα δυσκολότερη την οποιαδήποτε προσπάθεια χρήσης του παραγόμενου κώδικα σε οποιαδήποτε άλλη πλατφόρμα κλπ. Πιο συγκεκριμένα, κάνοντας behavioral simulation του παραγόμενου κώδικα από το HLS το οποίο γίνεται με τη βοήθεια του Vivado (για τη διασύνδεση και την παραγωγή των απαραίτητων file lists – βιβλιοθηκών) και χρησιμοποιώντας το modelsim για το ίδιο το simulation, αναλύοντας τα επιμέρους modules, παίρνουμε μηνύματα όπως τα ακόλουθα:

```
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 900 ps  Iteration: 14  Protected: /tb_system/sys/uNema/Rasterizer/i_Rasterizer_matmult_on
terize_ap_dmul_0_max_dsp_64_u/U0/<protected>/<protected>/<protected>/<protected>/<protected>/<protec
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 900 ps  Iteration: 14  Protected: /tb_system/sys/uNema/Rasterizer/i_Rasterizer_matmult_on
terize_ap_dmul_0_max_dsp_64_u/U0/<protected>/<protected>/<protected>/<protected>/<protected>/<protec
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 900 ps  Iteration: 14  Protected: /tb_system/sys/uNema/Rasterizer/i_Rasterizer_matmult_on
terize_ap_dmul_0_max_dsp_64_u/U0/<protected>/<protected>/<protected>/<protected>/<protected>/<protec
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 900 ps  Iteration: 14  Protected: /tb_system/sys/uNema/Rasterizer/i_Rasterizer_matmult_on
terize_ap_dmul_0_max_dsp_64_u/U0/<protected>/<protected>/<protected>/<protected>/<protected>/<protec
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 900 ps  Iteration: 16  Protected: /tb_system/sys/uNema/Rasterizer/i_Rasterizer_matmult_on
ted>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 900 ps  Iteration: 16  Protected: /tb_system/sys/uNema/Rasterizer/i_Rasterizer_matmult_on
ted>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>
# ** Warning: There is an 'U'|'X'|'W'|'Z'|'-' in an arithmetic operand, the result will be 'X'(es).
#    Time: 900 ps  Iteration: 16  Protected: /tb_system/sys/uNema/Rasterizer/i_Rasterizer_matmult_on
ted>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>/<protected>
```

Εικόνα 68: Questa protected warnings

Γεγονός που μας «ειδοποιεί» ότι υπάρχουν προστατευμένα κομμάτια ακόμα και στο simulation. Έτσι με μια μεγαλύτερη προσοχή στα waveform signals του modelsim:



Εικόνα 69: Modelsim encrypted waveform signals

Παρατηρούμε δηλαδή ότι ακόμα και σε επίπεδο RTL υπάρχουν κομμάτια που ακόμα και αν θέλουμε να αναλύσουμε τη λειτουργία τους (και ενδεχομένως να την αναπαράγουμε εκτός Xilinx FPGA) είναι αδύνατον. Αυτό γίνεται ακόμα πιο αντιληπτό σε περιπτώσεις όπως αυτής του netlist, όπου ολόκληρο το κύκλωμα περιγράφεται σε ένα μοναδικό αρχείο. Εκεί συναντούμε κομμάτια κώδικα όπως:

Εικόνα 70: Xilinx encrypted netlist

Εδώ βλέπουμε ότι υπάρχουν κομμάτια κώδικα, όπου είναι encrypted χρησιμοποιώντας AES encryption. Έτσι αν θεωρήσουμε ότι ήταν πρακτικά αδύνατον να κάνουμε “reverse

engineering” πλέον αντιλαμβανόμαστε ότι είναι και θεωρητικά αδύνατον. Αξίζει να σημειωθεί ότι το εργαλείο όπου θα θελήσουμε να κάνουμε simulation, πρέπει να είναι συμβατό και να έχει τα κλειδιά αποκωδικοποίησης αυτών των κομματιών ώστε να μπορέσει να λειτουργήσει. Είναι ένας πολύ σημαντικός λόγος που το Vivado και το Vivado HLS δεν είναι συμβατό με παλιότερες εκδόσεις λογισμικών. Επίσης αυτό σίγουρα δυσκολεύει το έργο ενδεχόμενων open source ή γενικά εναλλακτικών λύσεων πέραν τον επίσημα υποστηριζόμενων από την Xilinx. Απλά για εγκυκλοπαιδικούς και μόνο λόγους το netlist αρχείο της matmult είναι ~233000 γραμμές.

4.5 Integration

Ένα σημαντικό κομμάτι του χρόνου κατά την εκπόνηση της συγκεκριμένης διπλωματικής, καταναλώθηκε στην προσπάθεια ενσωμάτωσης του παραχθέντος κώδικα στην υπάρχουσα αλυσίδα εργαλείων (toolchain) της εταιρίας. Για τον λόγο αυτόν χρησιμοποιήθηκε ο συνδυασμός bash scripting καθώς και tcl scripting. Θα μπορούσε να επιτευχθεί μόνο με tcl scripting αλλά η tcl είναι πιο «δύσκολη» από το πολύ πιο ευρέως διαδεδομένο shell scripting. Η tcl στα περισσότερα hardware tools είναι μονόδρομος αφού όλες οι εντολές τους στο manual είναι σε tcl. Για τον καλύτερο έλεγχο του κώδικα κατά τη διάρκεια της ανάπτυξής του, χρησιμοποιήθηκε το git (versioning control).

Το Vivado, και το Vivado HLS κάθε φορά που εκτελούμε μια ενέργεια, εμφανίζει στη γραμμή εντολών του, την αντίστοιχη εντολή που χρησιμοποιήθηκε. Έτσι μπορούμε χωρίς πολύ κόπο να αντιγράψουμε τις εντολές αυτές σε ένα tcl script Συγκεκριμένα αφού επιλέξουμε ποιο fpga θέλουμε και την περίοδο και προσθέσουμε τα αρχεία μας και δοκιμάσουμε να συνθέσουμε το αρχικό μας tsi_rasterize, δημιουργείται το ακόλουθο tcl

```
open_project Rasterizer_HLS_2015_4
set_top tsi_rasterize
add_files ../hls_rasterizer/cleaned/rast_setup.cpp
add_files ../hls_rasterizer/cleaned/rast_traverse.cpp
add_files ../hls_rasterizer/cleaned/rast_vars.h
add_files ../hls_rasterizer/cleaned/tsi_rasterize.cpp
add_files -tb ../hls_rasterizer/simple_rasterizer_tb.cpp
open_solution "solution1"
set_part {xc7z020clg484-1}
create_clock -period 50 -name default
#source "./Rasterizer_HLS_2015_4/solution1/directives.tcl"
csim_design
csynth_design
cosim_design
export_design -format ip_catalog
```

Στην περίπτωση που θέλουμε να αυτοματοποιήσουμε τη σύνθεση, κάνουμε comment out (# στην αρχή της γραμμής) το csim (C simulation) και το cosim (C/RTL simulation).

Ομοίως συμβαίνει και για την matmult. Η ενδιάμεση FIFO που χρησιμοποιήθηκε, αποτελεί IP της Xilinx.

Για το Vivado, χρειάστηκε να δημιουργηθεί ένας wrapper σε Verilog, ώστε να συνδυάσει τα παραχθέντα IPs σε ένα κοινό project. Στον wrapper αυτόν εκτός από τη σύνδεση των IPs παράγονται και τα απαραίτητα σήματα ελέγχου για τα modules του HLS. Τα σήματα αυτά είναι:

ap_clk, ap_rst, ap_start, ap_done, ap_idle, ap_ready

Τα ανωτέρω σήματα υπάρχουν σε κάθε παραγόμενο από το HLS module.

Προφανώς εφόσον το σύστημα πρέπει να ενσωματωθεί στην υπόλοιπη gpu, τα σήματα αυτά δεν πρέπει να είναι ορατά έξω από τον rasterizer. Και αυτός είναι ένας λόγος της δημιουργίας του wrapper. Επίσης, στον wrapper κάνουμε assign τα σήματα εξόδου της matmult (Pixel [355:0]) στα αντίστοιχα σήματα εξόδου του rasterizer (x,y, colors κλπ). Η είσοδος και έξοδος της matmult αποφασίστηκε να είναι μία struct όπου περιέχονται όλα τα απαραίτητα στοιχεία του Pixel ώστε ο συγχρονισμός και η διαχείρισή τους να είναι ευκολότερη. Έτσι γνωρίζουμε πότε όλα τα στοιχεία είναι valid (κοινό σήμα valid) και ενημερώνουμε τα επόμενα modules της gpu.Έτσι πετυχαίνουμε το σύστημά μας να είναι 100% συμβατό με αυτό που αντικαθιστά.

Εφόσον λοιπόν ολοκληρώσαμε και τον wrapper, πρέπει όλο το σύστημα να ενσωματωθεί στο υπόλοιπο workflow. Αυτό γίνεται με τη χρήση bash scripting. Το Vivado μας δίνει τη δυνατότητα να επιλέξουμε να κάνει export τα scripts για τον υποστηριζόμενο από αυτό simulator (εδώ Questa). Τα scripts αυτά είναι 3: **_compile.do, **_elaborate.do και **_simulate.do. Το compile, περιέχει τα αρχεία που πρέπει να κάνουμε compile στον simulator, το elaborate κάνει link τις απαραίτητες βιβλιοθήκες της Xilinx και κάνει και το design optimize και το simulate το οποίο απλά καθορίζει το χρονικό διάστημα που θα «τρέξει» το simulation, τα σήματα που θα φαίνονται στον wave viewer κλπ. Η δουλειά μας λοιπόν είναι να προσθέσουμε τα αρχεία του compile.do ως αρχεία προς compilation της βασικής gpu και να κάνουμε link όλες τις βιβλιοθήκες που χρειάζονται και αναφέρονται στο elaborate, κατά την εκτέλεση του simulation. Στην συγκεκριμένη περίπτωση αυτό επιτυγχάνεται με την χρήση ενσωματωμένων στο *nix συστήματα εργαλείων όπως το sed (stream editor) και το cat ώστε απλά να αντιγράψουμε και να επικολλήσουμε τις γραμμές που χρειαζόμαστε στα αρχεία που θα γίνουν compile.

5. ΣΥΜΠΕΡΑΣΜΑΤΑ

Ο τομέας των γραφικών αποτελεί αδιαμφησβήτητα έναν από του πιο σημαντικούς και δύσκολους τομείς της επιστήμης των υπολογιστών (και όχι μόνο). Η απαίτηση για όλο και πιο ρεαλιστικά γραφικά, οδηγεί στην ανάγκη εξεύρεσης όλο και πιο πολλών εφικτών, από άποψη υπολογιστικού κόστους αλγορίθμων. Η ανάπτυξη κυρίως της τεχνολογίας των ημιαγωγών τα τελευταία χρόνια έχει βοηθήσει προς την δυνατότητα υλοποίησης αρκετά ρεαλιστικότερων γραφικών σε σύγκριση με το παρελθόν.

Το Vivado HLS ως εργαλείο μπορεί να βοηθήσει στην ταχεία ανάπτυξη αλγορίθμων, είτε γραφικών είτε διαφορετικού τομέα, όμως από τα αποτελέσματά μας, οδηγούμαστε στην αντίληψη ότι οι καλά σχεδιασμένες στο επίπεδο RTL υλοποιήσεις μπορούν να προσφέρουν πολύ καλύτερα αποτελέσματα.

Η έλευση συσκευών όπως είναι το Zynq οδηγεί στη δυνατότητα ταχύτατης κατασκευής υλικού που είναι συγκρινόμενο σε ταχύτητα και κατανάλωση με τα ASICs και σίγουρα πολύ πιο αποδοτικό από έναν επεξεργαστή γενικού σκοπού. Μάλιστα με την συνεργασία και του εργαλείου HLS μπορούν προγραμματιστές με ελάχιστη εμπειρία σε hardware να δημιουργήσουν ένα σχετικά γρήγορο επιταχυντή υλικού. Η χαμηλή τιμή και η αύξηση των επιδόσεων των νεότερων γενιών των FPGAs, τα έχει καταστήσει και ως πολλές φορές προτιμώμενες λύσεις σε εφαρμογές όπου είτε ο χρόνος εμφάνισης στην αγορά είναι πολύ σημαντικός είτε απαιτείται μικρός αριθμός μονάδων παραγωγής.

Παρόλα αυτά δεν δείχνουν να μπορούν τουλάχιστον ακόμα να συναγωνιστούν προϊόντα ευρείας κατανάλωσης, όπου ο συνδυασμός τιμής, ενέργειας και ταχύτητας είναι πολύ σημαντικός.

ΠΙΝΑΚΑΣ ΟΡΟΛΟΓΙΑΣ

Ξενόγλωσσος όρος	**Ελληνικός Όρος**
Bit	Διαδικό ψηφία
Resolution	Ανάλυση
Pixel	Εικονοστοιχείο
Color depth	Βάθος χρώματος
Color space	Χώρος χρωμάτων
True color	Πραγματικό χρώμα
Vector	Διάνυσμα
Thin lines	Λεπτές γραμμές
Curves	Καμπύλες
Fonts	Γραμματοσειρές
Logos	Λογότυπα
Rendering	Φωτοαπόδοση
Scene	Σκηνή
Geometry	Γεωμετρία
Texture	Υφή
Lighting	Φωτισμός
Shading	Σκίαση
Viewpoint	Οπτική γωνία
Equation	Εξίσωση
Real time	Πραγματικού χρόνου
Wireframe	Πλέγμα
Primitive	Αρχέτυπο
Format	Πρότυπο
Visibility	Ορατότητα
Hidden surface	Κρυμμένη επιφάνεια
Intersection point	Σημείο τομής
Loop	Επανάληψη
Perspective	Προοπτική
Projection	Προβολή
Bound	Όριο
Box	Κουτί
Integer	Ακέραιος
Floating point	Κινητής υποδιαστολής
Fixed point	Σταθερής υποδιαστολής
Frame	Πλαίσιο
Coverage	Επικάλυψη
Edge	Πλευρά
Function	Συνάρτηση
Attributes	Ιδιότητες
Barycentric	Βαρυκεντρικές
Coordinates	Συντεταγμένες
Weight	Βάρος
Area	Επιφάνεια
Painter	Ζωγράφος
Normal	Νόρμα
Linear	Γραμμικός
Interpolation	Παρεμβολή

Ray	Ακτίνα
Coefficients	Παράγοντες
Flow	Ροή
Design	Σχεδίαση
Develop	Αναπτύσσω
Constraints	Περιορισμοί
Directives	Ντιρεκτίβες
Efficient	Αποδοτικό
Precision	Ακρίβεια
Protocol	Πρωτόκολλο
Embedded system	Ενσωματωμένο σύστημα
FPGA	Διάταξη επαναπρογραμματιζόμενης λογική
Processing system	Σύστημα επεξεργασίας
Programmable logic	Προγραμματιζόμενη λογική
Bus	Δίαυλος
Abstract	Αφαιρετικός
Arbitration	Διαιτησία
Interface	Διεπαφή
Port	Θύρα
Interrupt	Διακοπή
Converter	Μετατροπέας
Analog	Αναλογικός
Digital	Ψηφιακός
Clock	Ρολόι
Interconnect	Διασύνδεση
Memory	Μνήμη
Map	Αντιστοίχιση
Simulation	Προσομοίωση
Generator	Δημιουργός
Tool	Εργαλείο
Embedded	Ενσωματωμένο
Standard	Πρότυπο
Implementation	Υλοποίηση
Switch	Διακόπτης
Matrix	Πίνακας
Performance	Απόδοση
Draw	Σχεδίαση
Rectangle	Ορθογώνιο
Latency	Καθυστέρηση
Shape	Σχήμα
Digital Signal Processing	Ψηφιακή επεξεργασία σήματος
Software	Λογισμικό
Hardware	Υλικό
Validation	Επαλήθευση

ΣΥΝΤΜΗΣΕΙΣ – ΑΡΚΤΙΚΟΛΕΞΑ – ΑΚΡΩΝΥΜΙΑ

FPGA	Field Programmable Gate Array
HLS	High Level Synthesis
RGB	Red Green Blue
RGBA	Red Green Blue Alf
FPS	Frames per second
GPU	Graphics Processor Unit
Fp	Floating point
API	Application Programming Interface
IO(B)s	Input/Output (Block)
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
ASIC	Application Specific Integrated Circuit
SoC	System on Chip
CLB	Configurable Logic Block
DSP	Digital Signal Processor
BUFG	Global buffer
BRAM	Block Random Access Memory
DRAM	Distributed Random Access Memory
AXI	Advanced Extensible Interface
LUT	Look-up Table
APU	Application Programmable Unit
PS	Programmable System
PI	Programmable Logic
FF	Flip Flop
VDS	Vivado Design SUite
SDK	Software Development Kit
TCL	Tool Command Language

ΑΝΑΦΟΡΕΣ

[1] http://vector-conversions.com/vectorizing/raster_vs_vector.html
[2] Κοροβέσης Χαράλαμπος, Φωτορεαλισμός και απόδοση, Χανιά ,2011
[3] https://en.wikipedia.org/wiki/Rendering_(computer_graphics)
[4] Scratchpixel.com "Rasterization: a Practical Implementation"
[5] Juan Pineda, A Parallel Algorithm for Polygon Rasterization August 1988
[6] Alois Zingl, "A Rasterizing Algorithm for Drawing Curves", Wien, 2012
[7] https://en.wikipedia.org/wiki/Ray_tracing_(graphics)
[8] https://en.wikipedia.org/wiki/Radiosity_(computer_graphics)
[9] Αθανάσιος Φίτσιος, Εφαρμογές Συνελικτικών Νευρωνικών Δικτύων με χρήση του Zynq-7000 SoC, Θεσσαλονίκη, Ιούλιος 2017
[10] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stewart, "The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All Programmable SoC", Department of Electronic and Electrical Engineering, University of Strathclyde, Glasgow, Scotland, UK, 1st Edition, 2014.
[11] Xilinx Inc, "ZC702 Evaluation Board for the Zynq-7000 XC7Z020 All Programmable SoC User Guide" UG850, v1.5, September 4, 2015.
[12] Xilinx Inc., *"Vivado Design Suite User Guide: Embedded Processor Hardware Design",* UG898, v. 2017.3, October 27, 2017
[13] Xilinx Inc, "Vivado Design Suite User Guide High-Level Synthesis", UG902 v2016.4, November 30, 2016
[14] Κεϊβάνη Ιωάννα ,"Υλοποίηση τεχνικών διαχείρισης του ρολογιού με στόχο την μείωση της κατανάλωσης ενέργειας σε πολυνηματικούς και πολυπύρηνους ενσωματωμένους επεξεργαστές γραφικών", Πάτρα, Ιανουάριος 2017