ΠΑΝΕΠΙΣΤΗΜΙΟ ΜΑΚΕΔΟΝΙΑΣ

ΠΡΟΓΡΑΜΜΑ ΜΕΤΑΠΤΥΧΙΑΚΩΝ ΣΠΟΥΔΩΝ

ΤΜΗΜΑΤΟΣ ΕΦΑΡΜΟΣΜΕΝΗΣ ΠΛΗΡΟΦΟΡΙΚΗΣ

Η ΣΥΝΕΡΓΑΣΙΑ ΤΩΝ ΤΕΧΝΟΛΟΓΙΩΝ OPENCL/OPENGL ΣΤΟ ΠΑΙΧΝΙΔΙ ΤΗΣ ΖΩΗΣ

Διπλωματική Εργασία

της

Νότα Μαρίας

Θεσσαλονίκη, Οκτώβριος 2018

Η ΣΥΝΕΡΓΑΣΙΑ ΤΩΝ ΤΕΧΝΟΛΟΓΙΩΝ OPENCL / OPENGL ΣΤΟ “ΠΑΙΧΝΙΔΙ ΤΗΣ ΖΩΗΣ”

Νότα Μαρία

Πτυχίο Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, 2009

Διπλωματική Εργασία

υποβαλλόμενη για τη μερική εκπλήρωση των απαιτήσεων του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΤΙΤΛΟΥ ΣΠΟΥΔΩΝ ΣΤΗΝ ΕΦΑΡΜΟΣΜΕΝΗ ΠΛΗΡΟΦΟΡΙΚΗ

Επιβλέπων Καθηγητής
Κωνσταντίνος Μαργαρίτης

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την

....

.....

Περίληψη

Η παρούσα εργασία έχει ως στόχο να αναδείξει την συνεργασία των τεχνολογιών OpenCL και OpenGL σε GPU, υλοποιώντας το παιχνίδι της ζωής του Conway (Conway's Game of Life). Οι υπολογιστική διεργασία της εφαρμογής στηρίζεται στη χρήση της OpenCL, ενώ η γραφική απεικόνιση του παιχνιδιού υλοποιείται με τη τεχνολογία OpenGL. Σκοπός της εργασίας είναι οι παραπάνω διεργασίες να επιτευχθούν με τη χρήση ενός κοινού αντικειμένου μνήμης που οι δύο τεχνολογίες θα μοιράζονται.

Στα κεφάλαια της εργασίας, αναλύεται τι είναι το παιχνίδι της ζωής, οι κανόνες του καθώς και γνωστά πρότυπα που παράγονται από τη συνεχή λειτουργία του. Στη συνέχεια, γίνεται αναφορά στα βασικά χαρακτηριστικά της κάθε τεχνολογίας και αναλύονται βασικές εντολές που χρησιμοποιήθηκαν στην εφαρμογή. Έπειτα, καθορίζουμε τις βασικές συνιστώσες που πρέπει να εφαρμοστούν ώστε να επιτευχθεί η συνεργασία των δυο API. Μετά το θεωρητικό πλαίσιο της εργασίας, αναλύονται τα στάδια υλοποίησης της εφαρμογής με τελικό στόχο την κινούμενη γραφική απεικόνιση της πορείας των κυττάρων. Η εργασία ολοκληρώνεται με τις μετρήσεις, συμπεράσματα και τις μελλοντικές κατευθύνσεις της.

Λέξεις Κλειδιά:

OpenCL, OpenGL, Συνεργασία OpenCL / OpenGL, παιχνίδι της ζωής, Game of Life

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Κωνσταντίνο Μαργαρίτη, όχι μόνο για την πολύτιμη βοήθεια του στην διάρκεια της διπλωματικής εργασίας αλλά και για τις σημαντικές γνώσεις που μου μετέδωσε στο ΠΜΣ. Επίσης, ευχαριστώ ιδιαίτερα τον Κώστα για την καθημερινή στήριξη σε όλη την διάρκεια των σπουδών μου και το Νίκο για τις χρήσιμες συμβουλές που μου έδωσε.

Περιεχόμενα

Πίνακας Εικόνων

Πίνακας Πινάκων

ΚΕΦΑΛΑΙΟ 1

1. Εισαγωγή

1.1 Πρόβλημα – Σημαντικότητα

Η παράλληλη επεξεργασία κερδίζει ολοένα και περισσότερο έδαφος στις μοντέρνες αρχιτεκτονικές των επεξεργαστών. Οι κάρτες γραφικών, από απλές συσκευές απεικόνισης γραφικών εξελίσσονται σε ταχύτατους επεξεργαστές με αποκλειστικό σκοπό την αύξηση της απόδοσης των εφαρμογών. Πολλές φορές, οι μονάδες επεξεργασίας γραφικών (GPU) καλούνται να επεξεργάζονται παράλληλα μεγάλους όγκους δεδομένων και ταυτόχρονα να απεικονίζουν τα δεδομένα αυτά στην οθόνη. Το Παιχνίδι της Ζωής του Conway είναι ένα πρόβλημα κατάλληλο για παράλληλη επεξεργασία, το οποίο απαιτεί ταυτόχρονα γραφική αναπαράσταση στην οθόνη στον ελάχιστο χρόνο, πρόβλημα κατάλληλο για κάρτες γραφικών.

1.2 Σκοπός της εργασίας

Σκοπός της εργασίας είναι να αναλυθούν οι τεχνολογίες OpenCL και OpenGL και να αποδοθεί η γραφική αναπαράσταση των δεδομένων στο παιχνίδι της ζωής του Conway μέσω της συνεργασίας τους.

1.3 Διάρθρωση της εργασίας

Στο $2^{ο}$ κεφάλαιο της εργασίας, αναλύεται το παιχνίδι της ζωής, οι κανόνες του καθώς και γνωστά πρότυπα που παράγονται εφαρμόζοντας τους κανόνες από γενιά σε γενιά. Επιπλέον, παρουσιάζονται οι παραλλαγές, η χρησιμότητα και οι πρακτικές εφαρμογές του παιχνιδιού στην πραγματική ζωή. Στη συνέχεια, στο $3^{ο}$ κεφάλαιο γίνεται αναφορά στα βασικά χαρακτηριστικά της OpenGL και αναλύονται βασικές εντολές που χρησιμοποιήθηκαν στην εφαρμογή. Επίσης, παρουσιάζεται το παιχνίδι της Ζωής με τη τεχνολογία OpenGL. Παρόμοια, στο κεφάλαιο 4, γίνεται μια εισαγωγή στην OpenCL και στη βασική ορολογία της. Το παιχνίδι της ζωής παρουσιάζεται με υπολογισμό OpenCL. Στο κεφάλαιο 5, καθορίζουμε τις βασικές συνιστώσες που πρέπει να εφαρμοστούν ώστε

να επιτευχθεί η συνεργασία των δυο API. Μετά το θεωρητικό πλαίσιο της εργασίας, στο κεφάλαιο 6 αναλύονται τα στάδια υλοποίησης της εφαρμογής με τελικό στόχο την κινούμενη γραφική απεικόνιση της πορείας των κυττάρων. Η εργασία ολοκληρώνεται στο κεφάλαιο 7 με τα αποτελέσματα και τις μετρήσεις και στο κεφάλαιο 8 με τα συμπεράσματα και τις μελλοντικές κατευθύνσεις της.

ΚΕΦΑΛΑΙΟ 2

2 Το παιχνίδι της ζωής

2.1 Εισαγωγή

Το παιχνίδι της ζωής είναι ένα κυτταρικό αυτόματο το οποίο κατασκευάστηκε από τον μαθηματικό καθηγητή του πανεπιστημίου Cambridge, John Horton Conway το 1970. Είναι ένα "παιχνίδι" το οποίο δεν αποτελείται από αληθινούς παίχτες, για αυτό και χαρακτηρίστηκε ως zero-player game. Δεν υπάρχει η έννοια της «νίκης» ή της «ήττας». Ουσιαστικά, το παιχνίδι προσομοιώνει την επιβίωση ή τον θάνατο κάποιου πληθυσμού ζωντανών κυττάρων (κυτταρικών αυτομάτων) που βρίσκονται στο χώρο και αλληλεπιδρούν μεταξύ τους. Η εξέλιξη του παιχνιδιού καθορίζεται μόνο από τις αρχικές συνθήκες, οι οποίες ορίζονται από τον χρήστη του παιχνιδιού και στη συνέχεια παρατηρεί τον τρόπο που αυτή εξελίσσεται. (1)

2.2 Κανόνες

Ο κόσμος του παιχνιδιού είναι ένας άπειρος ορθογώνιος πίνακας δύο διαστάσεων (πλέγμα) με τετράγωνα κελιά, που κάθε κελί μπορεί να βρίσκεται σε μία κατάσταση, «νεκρό» ή «ζωντανό». Κάθε κελί – ή αλλιώς κύτταρο - έχει ακριβώς 8 γείτονες που βρίσκονται οριζόντια, κατακόρυφα και διαγώνια δίπλα του. Σε κάθε χρονικό βήμα, το οποίο στη συνέχεια θα καλείται *γενιά*, παρατηρούνται οι ακόλουθες αλλαγές:

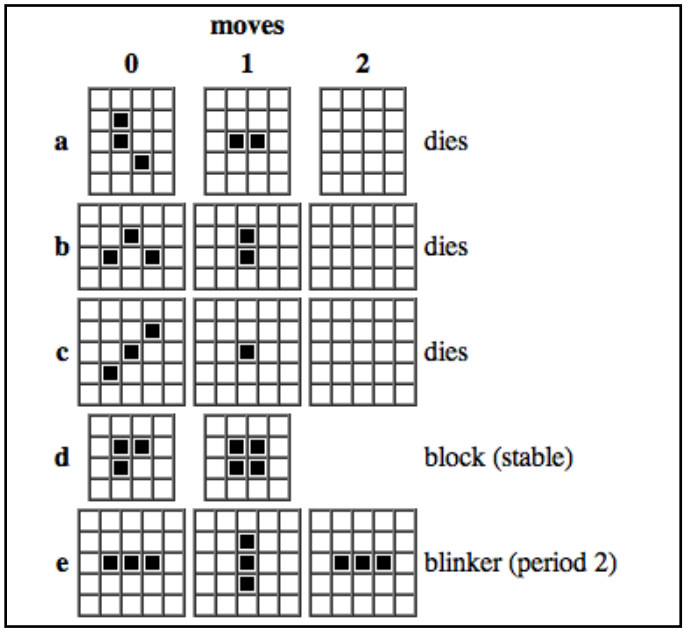
- Κάθε ζωντανό κελί με λιγότερους από δύο ζωντανούς γείτονες, πεθαίνει.
- Κάθε ζωντανό κελί με δύο ή τρεις ζωντανούς γείτονες επιβιώνει στην επόμενη γενεά.
- Κάθε ζωντανό κελί με περισσότερους από τρεις ζωντανούς γείτονες πεθαίνει.
- Κάθε νεκρό κελί με ακριβώς τρεις ζωντανούς γείτονες ζωντανεύει.

Το αρχικό μοτίβο αποτελεί τις αρχικές συνθήκες του συστήματος. Η πρώτη γενιά δημιουργείται εφαρμόζοντας τους κανόνες ταυτόχρονα σε όλα τα κύτταρα του συστήματος, καθώς άλλα γεννιούνται και άλλα πεθαίνουν ταυτόχρονα. Κάθε γενιά είναι συνάρτηση μόνο της προηγούμενης γενιάς. Στη συνέχεια, οι κανόνες εφαρμόζονται

επαναληπτικά σε κάθε νέα γενιά δημιουργώντας τις επόμενες. Συνοψίζοντας, η «ζωή» στο παιχνίδι περιλαμβάνει θεωρητικά 3 καταστάσεις:

- **Επιβίωση**: Ένα ζωντανό κύτταρο επιβιώνει όταν έχει δύο ή τρεις ζωντανούς γείτονες.
- **Θάνατος**: Ένα ζωντανό κύτταρο πεθαίνει όταν έχει είτε λιγότερους από δύο ζωντανούς γείτονες (μοναξιά) είτε περισσότερους από 3 ζωντανούς γείτονες (υπερπληθυσμό).
- **Γέννηση**: Ένα νεκρό κύτταρο γεννιέται όταν έχει ακριβώς τρείς ζωντανούς γείτονες. Ούτε λιγότερους ούτε περισσότερους.

Η μέτρηση των ζωντανών γειτόνων γίνεται ταυτόχρονα για όλα τα κύτταρα προτού εφαρμοστούν οι κανόνες. Με άλλα λόγια, οι εύρεση των αλλαγών στο σύστημα πρέπει να γίνει πριν την εφαρμογή των κανόνων στη γενιά, γεγονός που καθιστά αναγκαία τη χρήση υπολογιστή για την επεξεργασία της επόμενης γενιάς όλων του κυττάρων και μάλιστα ταυτόχρονα.



Εικόνα 2.1 – Παραδείγματα εξέλιξης των κυττάρων μετά από δύο γενεές

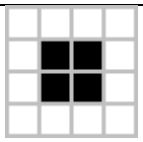
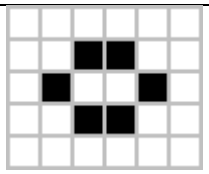
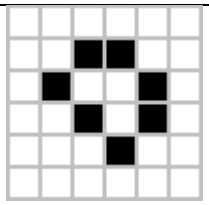
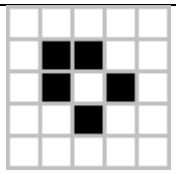
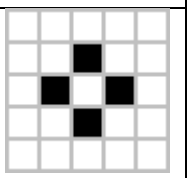
2.3 Πρότυπα

Καθώς οι παραπάνω κανόνες εφαρμόζονται επαναληπτικά, δημιουργούνται πολλές αλλαγές με το πέρασμα των γενεών, άλλοτε «όμορφες» δημιουργώντας σχήματα

και άλλοτε απροσδόκητες. Σε κάποιες περιπτώσεις, όλα τα κύτταρα εξαφανίζονται και η κοινωνία τελικά πεθαίνει (αν και αυτό μπορεί να συμβεί μετά από πολλές γενιές). Τα περισσότερα πρότυπα εκκίνησης όμως είτε φτάνουν σε σταθερούς αριθμούς ζωντανών κυττάρων (ο Conway τα αποκαλεί “still lifes”) είτε παραμένουν σε σχέδια τα οποία ταλαντεύονται για πάντα. Έτσι, με βάση την συμπεριφορά τους, τα πρότυπα χωρίζονται στις παρακάτω κατηγορίες:

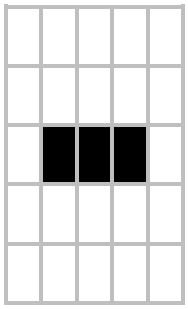
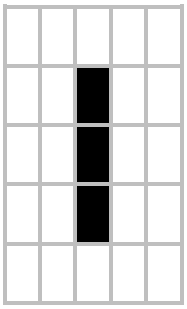
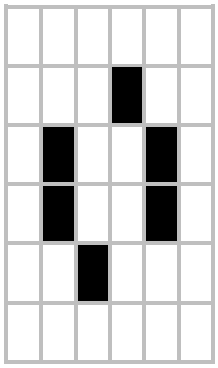
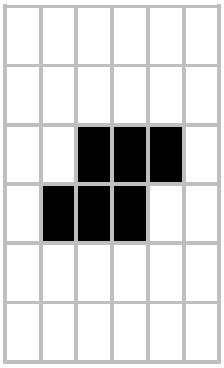
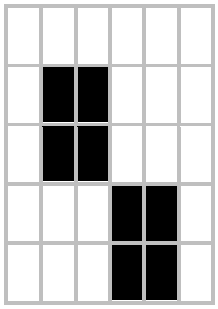
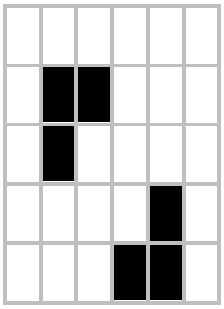
Ακίνητα (Still lifes): πρότυπα τα οποία παραμένουν ίδια από την μία γενιά στην άλλη. Το πιο γνωστό πρότυπο αυτής της κατηγορίας είναι το Block. Κάθε ζωντανό κύτταρο έχει ακριβώς τρεις γείτονες και δεν υπάρχει νεκρό κύτταρο που να έχει περισσότερους από δύο ζωντανούς γείτονες.

Πίνακας 2.1 – Σταθερά πρότυπα

Still lifes	
Block	
Beehive	
Loaf	
Boat	
Tub	

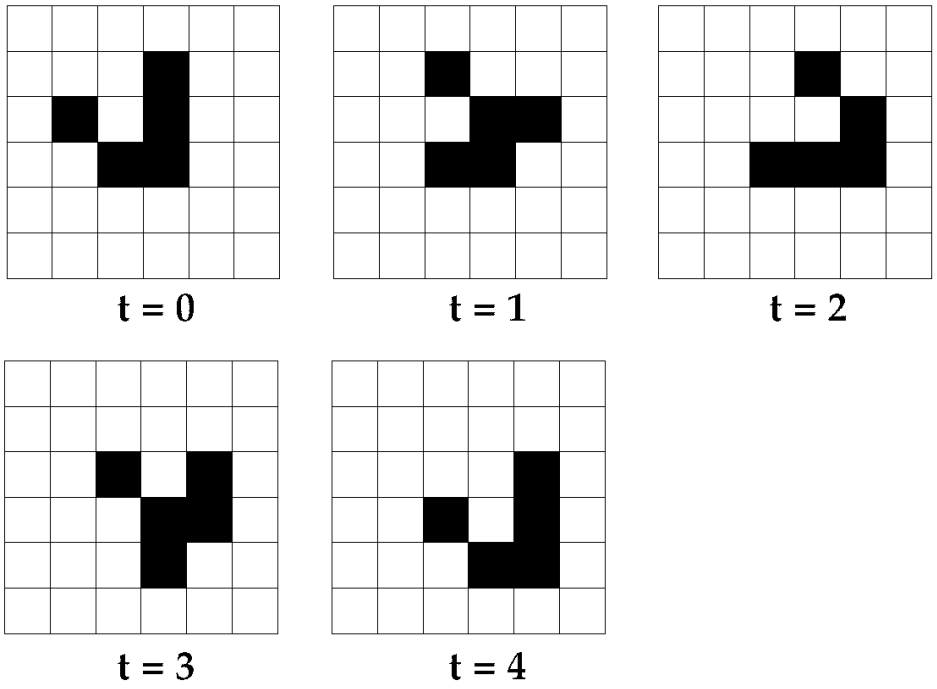
<u>Ταλαντούμενα (Oscillators)</u>: πρότυπα τα οποία αλλάζουν σε κάθε γενιά αλλά επιστρέφουν στην αρχική τους μορφή μετά από κάποιο αριθμό γενεών. Το πιο γνωστό πρότυπο αυτής της κατηγορίας είναι το Blinker που αποτελείται από τρία ζωντανά κύτταρα.

Πίνακας 2.2 – Επαναλαμβανόμενα πρότυπα

Oscillators	**1^{ο} βήμα**	**2^{ο} βήμα**
Blinker (period 2)		
Toad (period 2)		
Beacon (period 2)		

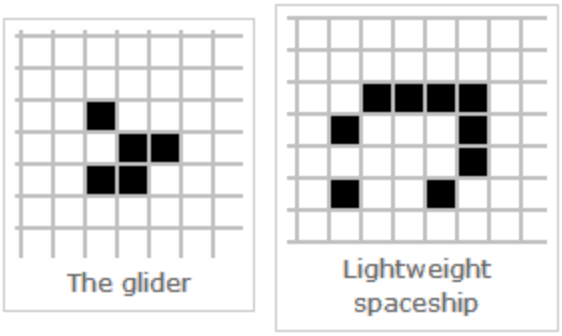
<u>Κινούμενα (Glider, Spaceships)</u>: πρότυπα τα οποία αλλάζουν σε κάθε γενιά και φαίνεται να μετακινούνται στο πλέγμα. Το πιο γνωστό πρότυπο αυτής της κατηγορίας είναι το Glider. Με την εφαρμογή των κανόνων, ύστερα από έναν αριθμό γενεών,

προκύπτει το αρχικό πρότυπο αλλά σε διαφορετική θέση. Έτσι, το πρότυπο φαίνεται να «κινείται».



Εικόνα 2.2 – Εφαρμογή κανόνων του παιχνιδιού στο πρότυπο Glider

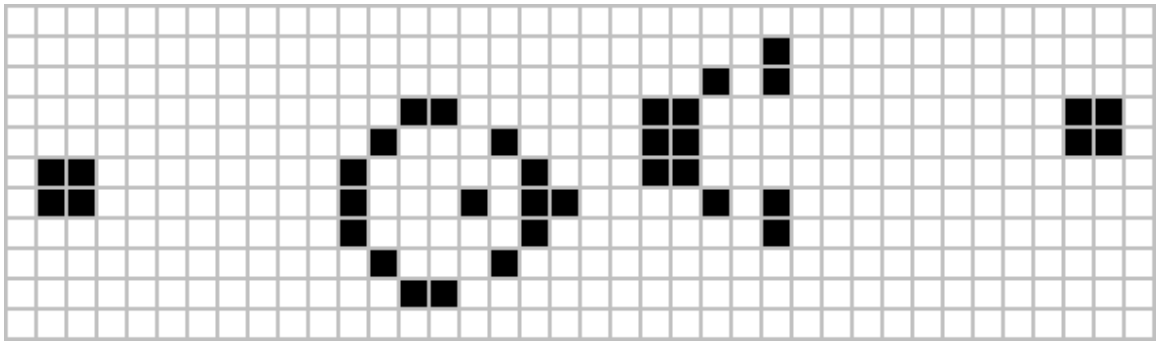
Ο Conway αρχικά υπέθεσε ότι κανένα μοτίβο δεν μπορεί να αυξάνεται συνεχώς, δηλαδή ότι για οποιαδήποτε αρχική διάταξη με πεπερασμένο αριθμό ζωντανών κυττάρων, ο πληθυσμός τους δεν θα μπορούσε να αναπτυχθεί πάνω από κάποιο άνω όριο. Στην αρχική εμφάνιση του παιχνιδιού στους "Μαθηματικούς Αγώνες", ο Conway προσέφερε βραβείο αξίας $50 στο πρώτο άτομο που θα μπορούσε να αποδείξει ή να διαψεύσει την εικασία πριν από το τέλος του 1970. Το βραβείο του Conway καταβλήθηκε σύντομα, όταν ανακαλύφθηκαν δύο διαφορετικοί τρόποι για την διάταξη ενός μοτίβου που ο πληθυσμός των ζωντανών κυττάρων μεγαλώνει για πάντα. Το Glider gun και το Puffer.



Εικόνα 2.3 - Απεικόνιση των προτύπων glider και Lightweight spaceship

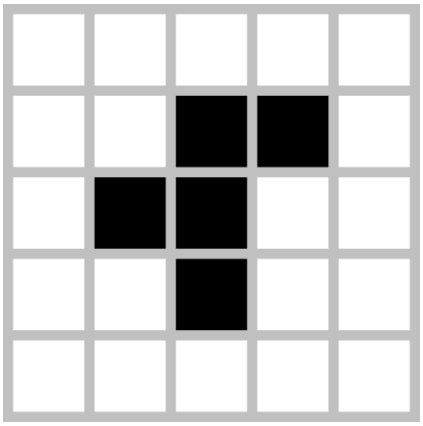
Αυξανόμενα πρότυπα: πρότυπα που αναπτύσσονται διαρκώς καθώς παράγουν νέα κύτταρα. Τα πιο ενδιαφέροντα είναι:

- Πυροβόλα Όπλα (Guns): επαναλαμβανόμενα πρότυπα που παράγουν ένα πρότυπο τύπου glider μετά από πεπερασμένο αριθμό γενεών. Το πιο απλό πρότυπο στην κατηγορία αυτή ανακαλύφθηκε το 1970, το Gosper glider gun, το οποίο παράγει ένα glider κάθε 30 γενιές. Το Gosper glider gun αποτελεί το πρώτο πρότυπο στο «Παιχνίδι της ζωής» που ο πληθυσμός των κυττάρων του αυξάνει συνεχώς. Από το 1970 και μετά, ερευνητές έχουν ανακαλύψει εκατοντάδες νέα πρότυπα αυτού του τύπου (2).



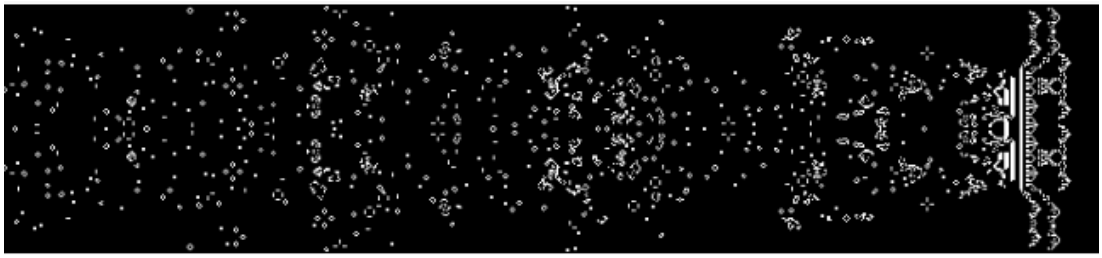
Εικόνα 2.4 – Απεικόνιση προτύπου Gosper glider gun

- R-pentomino: το μοναδικό πρότυπο το οποίο ξεκινώντας με πέντε κύτταρα καταλήγει να σταθεροποιηθεί μετά από 1103 γενιές (παρόμοια πρότυπα σταθεροποιούνται μετά από 10 γενιές). Είναι ένα ασταθή πρότυπο, με κάθε γενιά διαφορετική από την προηγούμενη, που στη διάρκεια της ζωής του περιέχει πολλά μοτίβα εκ των οποίων άλλα ταλαντεύονται και άλλα παραμένουν σταθερά. Η τελική μορφή του περιλαμβάνει 25 αντικείμενα, με τελικό πληθυσμό 116 ζωντανά κύτταρα.



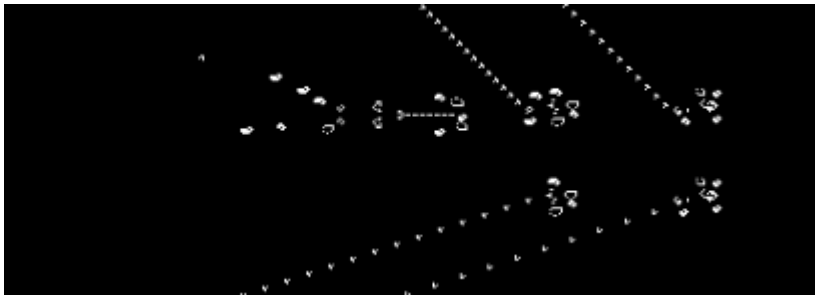
Εικόνα 2.5 - Απεικόνιση προτύπου R-pentomino

- Puffer trains: πρότυπα που μοιάζουν με κινούμενα τρένα. Αυτό που συμβαίνει είναι ότι το «τρένο» αρχίζει να κινείται και αφήνει πίσω του είτε σταθερά είτε ταλαντευόμενα «θραύσματα» ανά τακτά χρονικά διαστήματα.



Εικόνα 2.6 - Απεικόνιση προτύπου Puffer train

- Rakes: κινούμενα πρότυπα που εκπέμπουν gliders σε τακτά χρονικά διαστήματα καθώς κινούνται.

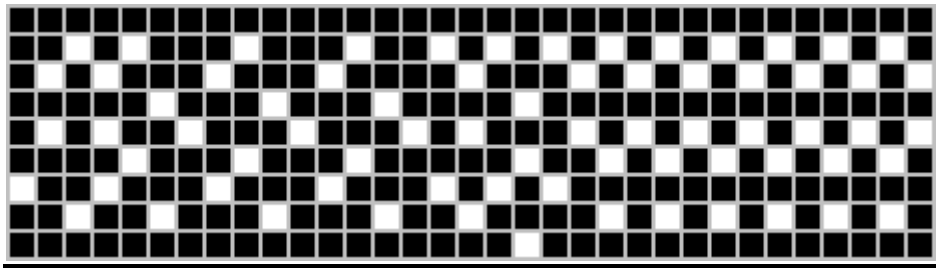


Εικόνα 2.7 – Απεικόνιση προτύπου Rakes

- Breeder: Πρόκειται για περίπλοκα ταλαντευόμενα μοτίβα που παράγουν «όπλα» σε τακτά χρονικά διαστήματα. Σε αντίθεση με τα άλλα πρότυπα, τα breeder αυξάνονται με τετραγωνικό ρυθμό ανάπτυξης. (2)

Εικόνα 2.8 - Απεικόνιση προτύπου Breeder

- Garden of Eden: πρότυπο που μπορεί να υπάρξει μόνο ως αρχικό πρότυπο. Με άλλα λόγια, κανένα μοτίβο δεν θα μπορούσε να παράγει αυτό το πρότυπο. (3)



Εικόνα 2.9 - Απεικόνιση προτύπου Garden of Eden

2.4 Παραλλαγές

Το παιχνίδι της ζωής σηματοδότησε την έναρξη για την δημιουργία νέων παρόμοιων κυτταρικών αυτομάτων. Με την παραλλαγή των κανόνων, δημιουργούνται νέες συμπεριφορές που μοιάζουν με το παιχνίδι της ζωής, αλλά στην πλειοψηφία τους είτε θα αναπτύσσονται χαοτικά είτε θα είναι πολύ «φτωχά» για να παρουσιάσουν ενδιαφέρον. Οι κανόνες του αρχικού παιχνιδιού συμβολίζονται με **B3/S23**, συμβολισμός που σημαίνει πως ένα κύτταρο γεννιέται (**B**irth) αν έχει ακριβώς 3 ζωντανούς γείτονες ή επιβιώνει (**S**urvives) αν έχει 2 ή 3 ζωντανούς γείτονες. Σε κάθε άλλη περίπτωση το κύτταρο πεθαίνει. Επομένως, ο συμβολισμός B6/S16 σημαίνει ότι ένα κύτταρο γεννιέται αν υπάρχουν έξι ζωντανοί γείτονες και επιβιώνει στην επόμενη γενιά αν υπάρχουν είτε ένας είτε έξι ζωντανοί γείτονες. Τα κυτταρικά αυτόματα που μπορούν να περιγραφούν με αυτόν τον τρόπο είναι γνωστά ως «Life-like» cellular automata. Γνωστή παραλλαγή του παιχνιδιού αποτελεί το HighLife, που περιγράφεται από τον κανόνα B36/S23 και είναι γνωστό για την συχνή εμφάνιση προτύπων που δημιουργούν κατ' επανάληψη αντίγραφα του εαυτού τους (replicators). (1)

Οι κανόνες του Conway μπορεί επίσης να αλλάξουν ως προς τον αριθμό των καταστάσεων, έτσι ώστε το παιχνίδι να γενικεύεται με 3 ή περισσότερες καταστάσεις. Η κάθε κατάσταση του κυττάρου αναπαρίσταται με διαφορετικό χρώμα κι έτσι προβάλλεται πολύχρωμο κυτταρικό αυτόματο. Γνωστή παραλλαγή σε αυτή τη κατηγορία αποτελεί το παιχνίδι «Immigration», στο οποίο τα ζωντανά κύτταρα αναπαρίστανται σε δύο χρώματα. Ένα κύτταρο που γεννιέται λαμβάνει το χρώμα που υπερισχύει ανάμεσα στους 3 γείτονες του. Αν και τα μοτίβα που δημιουργούνται στο Παιχνίδι «Immigration» δεν διαφέρουν από αυτά του παιχνιδιού της ζωής του Conway, ενδιαφέρον παρουσιάζει ο τρόπος με τον οποίο τα χρώματα αλληλεπιδρούν.

2.5 Χρησιμότητα

Η μελέτη και η ανάλυση του παιχνιδιού της ζωής, το οποίο αποτελεί παράδειγμα του μεγαλύτερου πεδίου κυτταρικών αυτομάτων, έχει αποφέρει πολλά οφέλη. Αυτά περιλαμβάνουν νέες εφαρμογές κυτταρικών αυτομάτων, όπως η ανάπτυξη καλύτερων μοντέλων ή συστημάτων προσομοίωσης στη βιολογία, τη χημεία και τη φυσική καθώς και στην αντιμετώπιση σύγχρονων προβλημάτων, όπως η κυκλοφοριακή συμφόρηση

στις πόλεις. Οι μελέτες σε αυτούς τους τομείς μπορούν να διαλευκάνουν μεγαλύτερα προβλήματα παρέχοντας νέες προοπτικές προσέγγισης και εξηγώντας επίσης τις περίπλοκες συμπεριφορές των κυττάρων, των ζώων και άλλων αντικειμένων. Για παράδειγμα:

- οι ιοί των υπολογιστών είναι επίσης σύστημα κυτταρικών αυτομάτων. Η εύρεση της θεραπείας για τους ιούς των υπολογιστών θα μπορούσε να προέρχεται από τα μοτίβα αυτού του απλού παιχνιδιού.
- Τα ανθρώπινα νοσήματα θα μπορούσαν να θεραπευτούν αν μπορούσαμε να καταλάβουμε καλύτερα γιατί τα κύτταρα ζουν και πεθαίνουν (4).

Το παιχνίδι της ζωής επίσης, ως κυτταρικό αυτόματο, μπορεί να λειτουργήσει όπως μία μηχανή Turing που υπολογίζει τιμές, ακολουθώντας τους απλούς κανόνες του Conway. Σε έρευνες αποδείχθηκε ότι τα κυτταρικά αυτόματα έχουν την ίδια υπολογιστική ισχύ με τις μηχανές Turing. Οι ακόλουθοι λόγοι οδηγούν τους ερευνητές να προσδιορίσουν ότι το Game of Life έχει όλες τις υπολογιστικές δυνατότητες των μηχανών Turing, πράγμα που σημαίνει ότι είναι **Turing πλήρης**:

- Οι μηχανές Turing έχουν την δυνατότητα να επαναλαμβάνονται απείρως. Επομένως, επειδή το Παιχνίδι της Ζωής έχει την ικανότητα να προσομοιώνει έναν άπειρο βρόχο ή άπειρη επανάληψη με τη μορφή ενός όπλου Gosper ή του puffer, οι άνθρωποι αποδέχτηκαν ότι το Game of Life έχει την υπολογιστική ικανότητα μιας μηχανής Turing.
- Μια συγκεκριμένη διαμόρφωση κυτταρικών αυτομάτων που ονομάζεται κανόνας 110 και χρησιμοποιεί ένα κυκλικό σύστημα είναι γνωστό ότι είναι Turing πλήρης. Μέσα από μια περίπλοκη μαθηματική απόδειξη, ο Stephen Wolfram και ο βοηθός του Matthew Cook απέδειξαν ότι ο κανόνας 110 είναι ικανό να υποστηρίξει καθολικούς υπολογισμούς.
- Ο κυτταρικός σχηματισμός ενός glider μπορεί να αλληλεπιδράσει με άλλα κυτταρικά πρότυπα και να δημιουργήσει λογικές πύλες, όπως τα AND, OR και NOT. Επομένως, είναι δυνατό να δημιουργηθεί ένα πρότυπο κυτταρικών αυτομάτων που λειτουργεί σαν μια μηχανή πεπερασμένων καταστάσεων με δύο μετρητές. Ο σχηματισμός που λειτουργεί σαν μια μηχανή πεπερασμένων

καταστάσεων έχει την ίδια υπολογιστική ισχύ με μια παγκόσμια μηχανή Turing, γνωστή και ως Turing πλήρης.

Από τα παραπάνω, προκύπτει ότι οι αλληλεπιδράσεις μεταξύ κυτταρικών αυτομάτων μπορούν να οδηγήσουν στην κατασκευή μηχανών Turing. (5)

2.6 Πρακτικές εφαρμογές

Τα κυτταρικά αυτόματα παρέχουν μεγάλη ποικιλία πρακτικών εφαρμογών στη πραγματική ζωή. Οι μελετητές σε διάφορους τομείς, όπως η επιστήμη των υπολογιστών, η φυσική , η βιολογία , η βιοχημεία , η οικονομία , τα μαθηματικά , η φιλοσοφία και οι γενετικές επιστήμες έχουν κάνει χρήση του τρόπου με τον οποίο μπορούν να προκύψουν πολύπλοκα σχέδια από την εφαρμογή των απλών κανόνων του παιχνιδιού. Μερικές από τις πρακτικές εφαρμογές είναι:

- **Κρυπτογραφία:** τα κυτταρικά αυτόματα μπορούν να χρησιμοποιηθούν για τον σχεδιασμό ενός κρυπτοσυστήματος δημόσιου κλειδιού. Οι αυτοματοποιημένοι κανόνες, οι οποίοι είναι αντιστρέψιμοι, είναι πρωταρχικοί υποψήφιοι για την αναστρέψιμη λειτουργία που απαιτείται για την κατασκευή ενός κρυπτοσυστήματος δημόσιου κλειδιού. Την ασφάλεια του συστήματος εγγυάται το γεγονός ότι είναι πολύ δαπανηρή διαδικασία παραγωγής της αρχικής κατάστασης από την τελική κρυπτογράφηση, χωρίς την γνώση των κανόνων που χρησιμοποιήθηκαν κατά την φάση της κρυπτογράφησης. (6)
- **Παραγωγή τυχαίων αριθμών:** τα κυτταρικά αυτόματα μπορούν να αποδώσουν τυχαίους ακέραιους αριθμούς. (7)
- **Παραγωγή μουσικών προτύπων:** Διάφορες τεχνικές μουσικής σύνθεσης χρησιμοποιούν το παιχνίδι της ζωής του Conway, ειδικότερα στη τεχνολογία MIDI. Με κατάλληλα προγράμματα λογισμικού, είναι δυνατή η δημιουργία ήχου από μοτίβα που παράγονται από τους κανόνες του παιχνιδιού.

2.7 Προγράμματα και online προσομοιωτές

Οι υπολογιστές χρησιμοποιηθήκαν και θεωρούνται πλέον αναγκαίοι για την παρακολούθηση των συνθέσεων του παιχνιδιού. Τα λογισμικά που αναπαριστούν το παιχνίδι περιλαμβάνουν ένα γραφικό περιβάλλον χρήστη για σχεδίαση προτύπων, την δυνατότητα εφαρμογής πολλαπλών κανόνων, καθώς και μια μεγάλη βιβλιοθήκη

σημαντικών προτύπων του παιχνιδιού. Ένα από τα δημοφιλέστερα λογισμικά ανοιχτού κώδικα για την προσομοίωση του παιχνιδιού είναι το “Golly”, το οποίο προσομοιώνει και άλλα κυτταρικά αυτόματα και χρησιμοποιεί τον αλγόριθμό «HashLife» για εξαιρετικά γρήγορο υπολογισμό. Επίσης, υπάρχουν πολλές σελίδες στο διαδίκτυο που παρέχουν online προσομοιωτή του παιχνιδιού. Μερικά από αυτά είναι:

- https://bitstorm.org/gameoflife/
- http://web.mit.edu/jb16/www/6170/gameoflife/gol.html
- https://pmav.eu/stuff/javascript-game-of-life-v3.1.1/

Τα παραπάνω links, προσομοιώνουν το παιχνίδι της ζωής για διαφορετικές αρχικές συνθήκες με χρήση απλής γραφικής διασύνδεσης.

ΚΕΦΑΛΑΙΟ 3

3 OpenGL

3.1 Εισαγωγή

Το OpenGL (Open Graphics Library) είναι ένα ανεξάρτητο από πλατφόρμες πρότυπο API για την παραγωγή 3D (συμπεριλαμβανομένων 2D) γραφικών. Είναι μια Διασύνδεση Προγραμματισμού Εφαρμογών (Application Programming Interface - API) που εμπεριέχει το σύνολο όλων των συναρτήσεων που πρέπει να υλοποιεί μια βιβλιοθήκη γραφικών για να πετύχει γραφική αναπαράσταση. Χρησιμοποιείται ανεξαρτήτου γλώσσας προγραμματισμού. Το πρότυπο OpenGL είναι μια διασύνδεση του λογισμικού με την κάρτα γραφικών. Με άλλα λόγια, οι εφαρμογές που χρησιμοποιούν γραφική απεικόνιση, διαθέτουν εντολές που μπορούν να κατευθύνονται απευθείας στην κάρτα γραφικών και να την επιταχύνουν.

3.2 Ιστορικά στοιχεία

Το OpenGL προέρχεται από το σύστημα GL ("Graphics Library") που εφευρέθηκε από την εταιρία Silicon Graphics Inc ως λύση για τον προγραμματισμό γραφικών εφαρμογών σε πολύ απαιτητικό επίπεδο. Με το πέρασμα των χρόνων, το ενδιαφέρον για τη μεταφορά του συστήματος GL σε διαφορετικές συσκευές ολοένα και αυξανόταν. Έτσι, το 1992 ανακοινώθηκε μια παραλλαγή του GL - που ονομάζεται OpenGL. Σε αντίθεση με το αρχικό σύστημα GL, η τεχνολογία OpenGL σχεδιάστηκε ώστε να είναι ανεξαρτήτου πλατφόρμας και να λειτουργεί ανεξάρτητα από το υλικό του υπολογιστή. Ο συνδυασμός της δύναμης και της φορητότητας του OpenGL οδήγησε στην γρήγορη αποδοχή του ως πρότυπο για προγραμματισμό γραφικών υπολογιστών (8) . Σήμερα, το OpenGL αναπτύσσεται από την OpenGL working group (www.opengl.org) της εταιρίας Khronos , που έχει τον έλεγχο της τεχνολογίας από το 2006. (9)

Version and year	Major functionality changes
OpenGL 1.0, 1992	The first version, with fixed functionality rendering pipeline.
OpenGL 1.1, 1997	Extensions including improved pixel blending, texture effects.
OpenGL 2.0, 2004	Introduction of the programmable rendering pipeline, using GLSL, the OpenGL shading language.
OpenGL 3.0, 2008	Fixed function pipeline deprecated.
OpenGL 4.0, 2010	Changes to functionality to allow OpenGL programs access to hardware features designed for Direct3D.

Εικόνα 3.1 - Οι σημαντικότερες εκδόσεις της OpenGL (8)

3.3 Βασικές λειτουργίες της OpenGL

Το OpenGL είναι ένα πρότυπο αναπαράστασης γραφικών που μπορεί να υλοποιηθεί σε οποιαδήποτε γλώσσα προγραμματισμού. Το πρώτο στάδιο, λοιπόν, είναι να καθοριστεί η γλώσσα προγραμματισμού της εφαρμογής. Υπάρχουν έτοιμες βιβλιοθήκες του OpenGL, τα λεγόμενα bindings (συνδέσεις), σε πολλές γλώσσες, όπως η C#, Java, Python κ.α. οι οποίες διευκολύνουν την ανάπτυξη λογισμικού με χρήση της OpenGL. Όλες οι υλοποιήσεις της παρούσας εργασίας βασίστηκαν στα bindings των γλωσσών C/C++ (10). Παρακάτω, περιγράφονται οι βασικές λειτουργίες που θα μπορούσε να αναπαραστήσει η OpenGL σε μια εικόνα:

- Παρέχει 3D γεωμετρικά σχήματα, όπως γραμμές, πολύγωνα, τρίγωνα, σφαίρες, κύβους, τετραγωνικές επιφάνειες και καμπύλες NURBS.
- Παρέχει μετασχηματισμούς 3D μοντελοποίησης και λειτουργίες προβολής για τη προβολή τρισδιάστατων σκηνών χρησιμοποιώντας την ιδέα μιας εικονικής κάμερας.
- Αποδίδει υψηλής ποιότητας σκηνές που συμπεριλαμβάνουν κρυφές επιφάνειες, πολλαπλές πηγές φωτός, αναπαράσταση υλικών, διαφάνειας, υφής, θολώματος.
- Υποστηρίζει τον χειρισμό των εικόνων ως pixels, επιτρέποντας εφέ, όπως θαμπάδα, βάθος πεδίου και σκίαση. (8)

3.4 Βιβλιοθήκες

Βασική βιβλιοθήκη (*OpenGL core library*): Η βασική βιβλιοθήκη του OpenGL περιέχει τις κύριες εντολές σχεδίασης. Όλες οι εντολές της βιβλιοθήκης αυτής διακρίνονται από το πρόθεμα ***gl***. Πολλές από τις συναρτήσεις της δέχονται προκαθορισμένα ορίσματα (συμβολικές σταθερές), τα οποία έχουν οριστεί στη βιβλιοθήκη και αντιστοιχούν σε διάφορες παραμέτρους ή καταστάσεις λειτουργίας. Κατά σύμβαση, οι σταθερές αυτές ξεκινούν με το πρόθεμα GL_ . (11)

OpenGL Utility Library (GLU): Περιλαμβάνει συναρτήσεις για τη δημιουργία πιο σύνθετων σχημάτων, όπως σύνθετες καμπύλες και επιφάνειες. Κάθε υλοποίηση με OpenGL εμπεριέχει τη βιβλιοθήκη GLU και όλες οι εντολές της βιβλιοθήκης GLU ξεκινούν με το πρόθεμα ***glu***.

OpenGL Utility Toolkit (GLUT): Η βιβλιοθήκη αυτή περιλαμβάνει εντολές απεικόνισης παραθύρων στην οθόνη, δημιουργίας menus, διαχείρισης γεγονότων κλπ. Όλες οι εντολές της ξεκινούν με το πρόθεμα ***glut***. Η βιβλιοθήκη GLUT περιέχει εντολές εισόδου-εξόδου και πετυχαίνει την αλληλεπίδραση του προγραμματιστή με την εφαρμογή.

3.5 Βασικοί τύποι δεδομένων του OpenGL

Οι τύποι δεδομένων του OpenGL είναι όμοιοι με τους αντίστοιχους που ορίζονται στη γλώσσα C. Παρακάτω, παρουσιάζεται η αντιστοιχία των βασικών τύπων του OpenGL με αυτών της C. Το OpenGL χρησιμοποιεί τους δικούς του τύπους δεδομένων ώστε να διατηρεί την φορητότητά του και η εφαρμογή να εκτελείται ανεξάρτητα από το υλικό του υπολογιστή.

Πίνακας 3.1 – Οι Βασικοί Τύποι Δεδομένων της OpenGL (12)

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	signed short	GLshort
i	32-bit integer	int	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int	GLuint, GLenum, GLbitfield

3.6 Βασικές εντολές

Για την δημιουργία παραθύρων είναι απαραίτητες οι παρακάτω συναρτήσεις:

3.6.1 void glutInit (int **argc*, char ***argv*)

Η συνάρτηση glutInit() αρχικοποιεί τη βιβλιοθήκη GLUT . Πρέπει να καλείται πριν από οποιαδήποτε άλλη συνάρτηση της GLUT. Η glutInit() επίσης, προσφέρει την δυνατότητα επιλογών από την γραμμή εντολών. Μερικές επιλογές για το παράθυρο είναι -iconic, -geometry, -display. Οι παράμετροι argc και argv είναι το πλήθος και οι τιμές των ορισμάτων που περνάμε στο πρόγραμμα μέσω της γραμμής εντολών.

3.6.2 int glutCreateWindow(char **name*)

Η συνάρτηση glutCreateWindow() ανοίγει ένα παράθυρο. Τα χαρακτηριστικά που θα έχει το παράθυρο (όπως πλάτος, ύψος, θέση) πρέπει να έχουν οριστεί προηγουμένως με τις κατάλληλες συναρτήσεις. Η παράμετρος name είναι το όνομα που θα εμφανίζεται ως τίτλος του παραθύρου. Το παράθυρο δεν εμφανίζεται μέχρι να εκτελεστεί πρώτα η εντολή glutMainLoop(). Η τιμή που επιστρέφει η συνάρτηση είναι ένας ακέραιος που ταυτοποιεί μοναδικά το παράθυρο (window ID) και χρησιμοποιείται για τον έλεγχο του παραθύρου σε περίπτωση που η εφαρμογή εμφανίζει πολλά παράθυρα.

3.6.3 void glutInitWindowSize(int *width*, int *height*)

Η συνάρτηση glutInitWindowSize() θέτει το αρχικό μέγεθος του παραθύρου. Οι παράμετροι width και height καθορίζουν το πλάτος και το ύψος του σε pixels. Μπορεί να

μεταβληθεί οποιαδήποτε στιγμή μέσα στο πρόγραμμα από τον χρήστη με την χρήση του ποντικιού.

3.6.4 void glutInitWindowPosition(int *x*, int *y*)

Η συνάρτηση glutInitWindowPosition() καθορίζει την αρχική θέση του παραθύρου σε σχέση με την επάνω αριστερή γωνία του οθόνης. Οι παράμετροι x και y καθορίζουν την απόσταση σε pixels.

3.6.5 void glutInitDisplayMode(unsigned int *mode*)

Η συνάρτηση glutInitDisplayMode() καθορίζει τον τύπο των γραφικών και το χρωματικό μοντέλο στο παράθυρο που δημιουργήθηκε με την glutCreateWindow(). Η παράμετρος mode μπορεί να πάρει τις παρακάτω σταθερές τιμές του OpenGL: GLUT_INDEX, GLUT_ALPHA, GLUT_DEPTH, GLUT_STENCIL, GLUT_MULTISAMPLE, GLUT_STEREO, GLUT_SINGLE, GLUT_DOUBLE, GLUT_RGB, GLUT_RGBA.

GLUT_SINGLE: Τεχνική της απλής ενταμίευσης (χρησιμοποιείται ένας buffer χρωματικών τιμών). Είναι κατάλληλη για τη σχεδίαση στατικών σκηνών.

GLUT_DOUBLE: Τεχνική της διπλής ενταμίευσης (double buffering) χρησιμοποιούνται δηλαδή δύο buffer χρωματικών τιμών. Η επιλογή αυτή ενδείκνυται για την παρουσίαση κινούμενων γραφικών.

GLUT_RGB: Περιγράφει το χρωματικό μοντέλο RGB. Κάθε χρώμα περιγράφεται από τους συντελεστές βάρους της κόκκινης (Red), της πράσινης (Green) και μπλε (Bleu) χρωματικής συνιστώσας του.

GLUT_RGBA: Περιγράφει το χρωματικό μοντέλο RGBA. Κάθε χρώμα περιγράφεται από τους συντελεστές βάρους της κόκκινης (Red), της πράσινης (Green) και μπλε (Bleu) χρωματικής συνιστώσας του. Το (Alpha) χρησιμοποιείται συνήθως για την σκίαση.

Περισσότερες πληροφορίες για τις παραμέτρους υπάρχουν στην επίσημη ιστοσελίδα του OpenGL (13)

3.6.6 void glutSetOption (GLenum eWhat, int value)

Η συνάρτηση glutSetOption() αποθηκεύει την τιμή της παραμέτρου value στην μεταβλητή κατάστασης που ορίζεται στην παράμετρο eWhat. Για παράδειγμα, η

παράμετρος eWhat μπορεί να πάρει προκαθορισμένες τιμές, όπως η GLUT_ACTION_ON_WINDOW_CLOSE η οποία ελέγχει τι συμβαίνει όταν ένα παράθυρο κλείνει από το χρήστη ή το σύστημα. Η τιμή value μπορεί να πάρει τιμές, όπως GLUT_ACTION_CONTINUE_EXECUTION, η οποία συνεχίζει η εκτέλεση των άλλων παραθύρων, GLUT_ACTION_EXIT, η οποία αμέσως τερματίζει την εφαρμογή, GLUT_ACTION_GLUTMAINLOOP_RETURNS, όπου φεύγει από την δομή επανάληψης της GlutMainLoop. (14)

3.6.7 void glFlush (void)

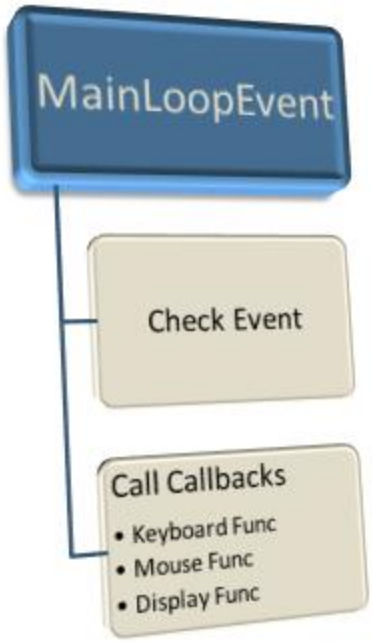
Η εντολή glFlush() εξαναγκάζει την εκτέλεση των εντολών που εκκρεμούν. Ο σκοπός της συνάρτησης είναι να καθοδηγήσει το OpenGL, ώστε να ανανεώνει σωστά την εικόνα στην οθόνη. Αναγκάζει τα περιεχόμενα των εσωτερικών buffer του OpenGL να εμφανίζονται στην οθόνη. Αν δεν υπάρχει η glFlush() στο τέλος των εντολών του OpenGL, τότε δεν υπάρχει εγγύηση ότι στην οθόνη θα εμφανιστεί η σωστή ενημερωμένη εικόνα. (8)

3.6.8 void glutDisplayFunc (*void (*func)(void)*)

Η συνάρτηση glutDisplayFunc() εκτελείται κάθε φορά που η εφαρμογή διαπιστώσει ότι απαιτείται επανασχεδιασμός της εικόνας. Ως όρισμα δέχεται μια συνάρτηση τύπου void, στην οποία εμπεριέχεται ο κώδικας σχεδίασης των γραφικών της οθόνης. Εντάσσεται στις συναρτήσεις της βιβλιοθήκης GLUT που καλούνται συναρτήσεις κλήσης και η χρήση της είναι υποχρεωτική σε κάθε εφαρμογή με OpenGL.

3.6.9 void glutMainLoop (void)

Η συνάρτηση glutMainLoop() ενεργοποιεί τον κύκλο διαχείρισης γεγονότων (event processing loop). Στον κύκλο αυτό, η εφαρμογή αναμένει επ' άπειρον και ανταποκρίνεται σε γεγονότα (Εικόνα 3.2), όπως λ.χ. στο πάτημα ενός κουμπιού, στην αλλαγή του σκηνικού ή στην κίνηση του ποντικιού. Εμπεριέχεται στη βιβλιοθήκη GLUT και μόλις ξεκινήσει να εκτελείται, ο βρόχος αυτός θα συνεχίσει για όσο διάστημα εκτελείται και το πρόγραμμα (11).



Εικόνα 3.2 – Γραφική αναπαράσταση της εντολής glutMainLoop()

Κάθε φορά, η βιβλιοθήκη GLUT ελέγχει αν υπάρχουν αλλαγές από την τελευταία κλήση. Αν υπάρχει, καλεί τις κατάλληλες call back functions. Σε ψευδοκώδικα, η λειτουργία της glutMainLoop() φαίνεται στην Εικόνα 3.3 (8).

```
while (1) { /* loop forever */
  if (the application has changed the graphics) {
     call the DISPLAY callback function;
     }

  if (the window has been moved or resized) {
     call the RESHAPE callback function;
     }

  if (any keyboard and/or mouse events have happened) {
     call the KEYBOARD and/or MOUSE callback function;
     }

  call the IDLE callback function;

} /* while */
```

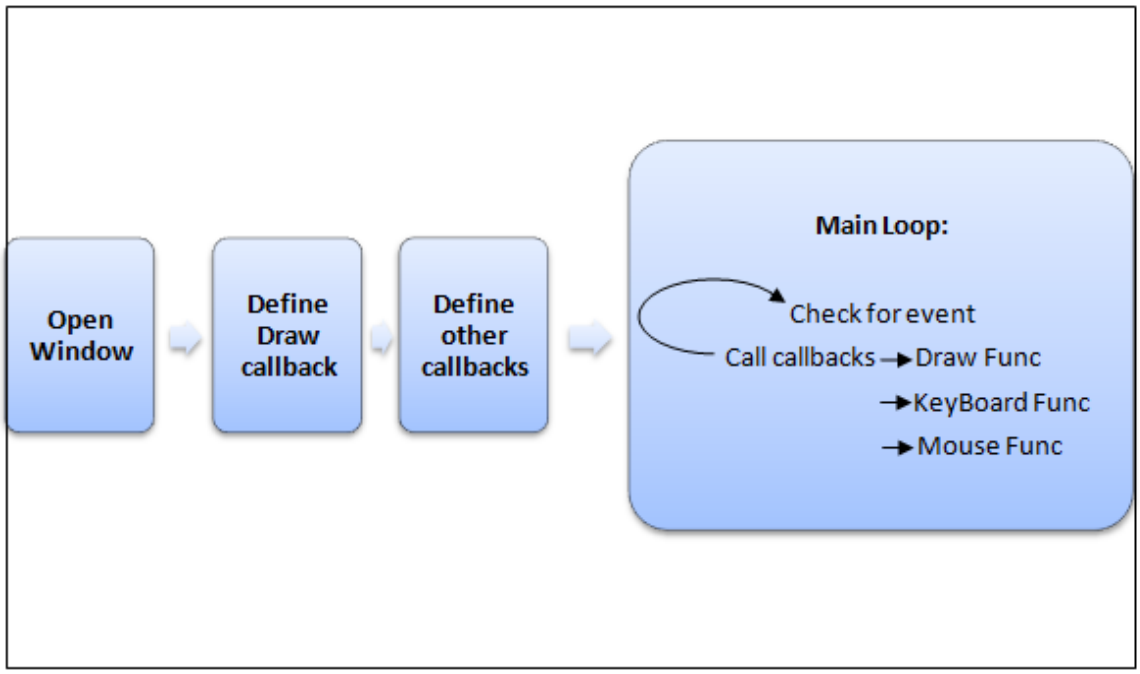
Εικόνα 3.3 - Ψευδοκώδικας με τη λειτουργία της εντολής glutMainLoop()

3.6.10 void glutIdleFunc (*void (*func)(void)*)

Η συνάρτηση glutIdleFunc() ανήκει στις εντολές της βιβλιοθήκης GLUT. Ορίζεται έτσι ώστε ένα πρόγραμμα να μπορεί να επεξεργάζεται εντολές στο παρασκήνιο ή να προβάλει κινούμενα γραφικά όταν δεν λαμβάνονται συμβάντα από το σύστημα, τα γνωστά events. Ως όρισμα, δέχεται μια συνάρτηση τύπου void στην οποία εμπεριέχεται ο κώδικας σχεδίασης των γραφικών, όταν δεν λαμβάνονται συμβάντα. Όταν το όρισμα είναι NULL, απενεργοποιείται η λειτουργία της.

3.6.11 void glutPostRedisplay (void)

Η συνάρτηση glutPostRedisplay() ανήκει στις εντολές της βιβλιοθήκης GLUT. Καλείται όταν απαιτείται επανασχεδιασμός της οθόνης. Το OpenGL καλεί την συνάρτηση που περάστηκε ως παράμετρος στην glutMainLoop() (συνήθως ονομάζεται display) και στην επόμενη επανάληψη του κύκλου διαχείρισης των γεγονότων (glutMainLoop) επανασχεδιάζει την οθόνη με τις νέες ρυθμίσεις.



Εικόνα 3.4 - Περιγραφή λειτουργίας του GLUT framework

3.7 Το Παιχνίδι της Ζωής με OpenGL

Παρακάτω παρουσιάζεται η υλοποίηση του Game of Life με υπολογισμό και γραφική αναπαράσταση με OpenGL. Ξεκινώντας με τη main(), ορίζεται τυχαία η αρχική κατάσταση των κυττάρων με την συνάρτηση rand(). Στη συνέχεια, αρχικοποιείται η βιβλιοθήκη GLUT με την glutInit() και παραμετροποιείται το παράθυρο προβολής με τις συναρτήσεις glutInitWindowSize(), glutInitWindowPosition(), glutCreateWindow(), όπως φαίνεται στον κώδικα παρακάτω. Τέλος, καλείται η glutMainLoop(), ώστε να ενεργοποιηθεί ο κύκλος διαχείρισης γεγονότων. (15)

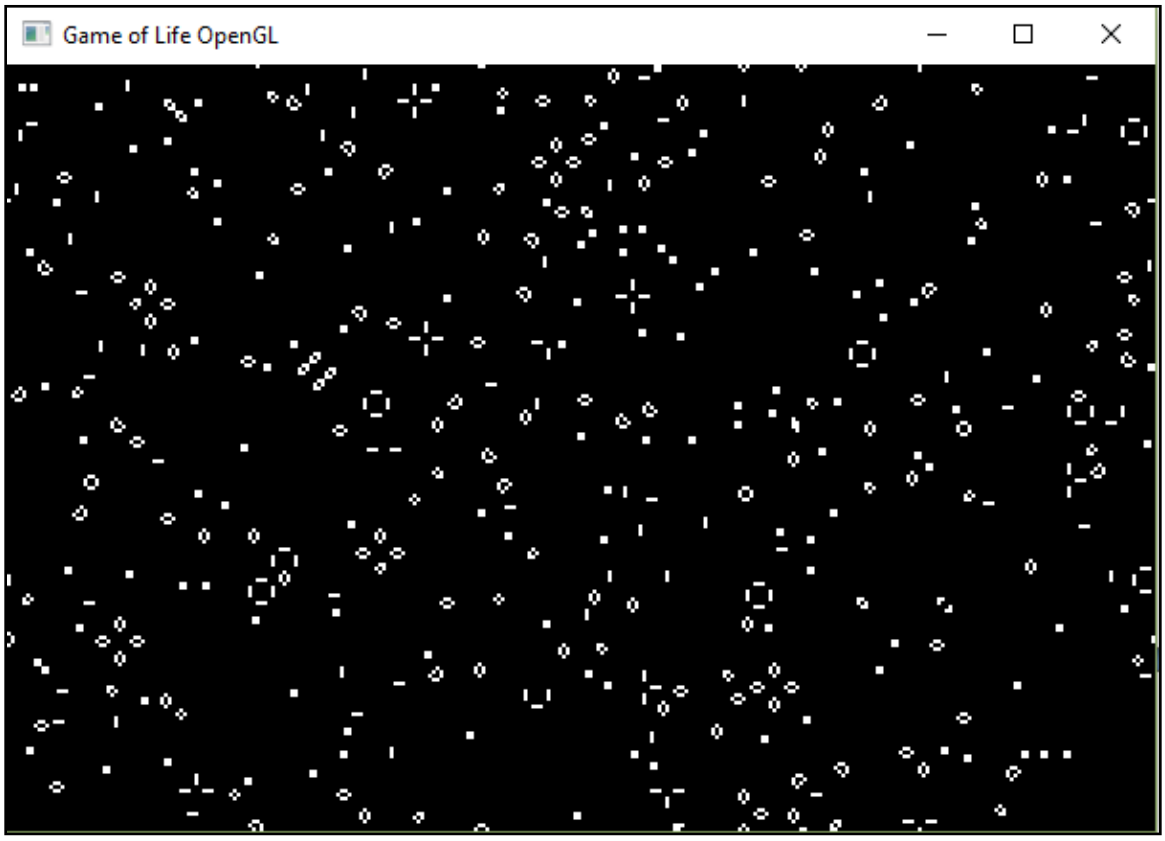
```
int main(int argc, char **argv)
{
	for (int x = 0; x < X; x++)
		for (int y = 0; y < Y; y++)
		{
			p[x][y].life = rand() % 2;
			p[x][y].next = 0;
		}
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
	glutInitWindowSize(w, h);
	glutInitWindowPosition(-10, -10);
	glutCreateWindow("Game of Life OpenGL");
```

```
	glClearColor(0, 0, 0, 1.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(0, w, h, 0, -1, 1);
	glutDisplayFunc(display);
	timer();
	glutMouseFunc(mouse);
	glutMotionFunc(motion);
	glutPassiveMotionFunc(motionpass);
	mousewrite();
	glutMainLoop();
}
```

Η δομή που χρησιμοποιείται στον κώδικα είναι ένα structure που διατηρεί στοιχεία για τη τρέχουσα και την επόμενη κατάσταση του κυττάρου. Η συνάρτηση display() περιέχει τον κώδικα σχεδίασης των γραφικών της οθόνης. Με την glBegin(GL_POINTS) δηλώνεται ότι ακολουθεί η σχεδίαση σημείων και με την glVertex2f() καθορίζονται οι κορυφές οι οποίες θα σχεδιαστούν. Η σχεδίαση των σημείων τελειώνει με την glEnd().

```
struct P
{
	bool life; // true if cell is alive
	int next; // next state of cell
} p[X][Y];

void display()
{
	glClear(GL_COLOR_BUFFER_BIT);
	glColor3ub(255, 255, 255);
	glPointSize(size_);
	glBegin(GL_POINTS);
	for (int y = 0; y < Y; ++y)
		for (int x = 0; x < X; ++x)
		if (p[x][y].life) glVertex2f(size_ / 2 + x*size_, size_ / 2 + y*size_);

	if (m_down && m_x > 0 && m_y > 0 && m_x < X * size_ && m_y < Y*size_)

		p[m_x / size_][m_y / size_].life = 1;

	else
	{
		int x = m_x / size_;  int y = m_y / size_;
		glVertex2f(size_ / 2 + size_ * x, size_ / 2 + size_ * y);
	}

	glEnd();
	glutSwapBuffers();

}
```

Ο κώδικας υλοποιήσης παρατίθεται ολοκληρωμένος στο παράρτημα της παρούσας εργασίας. Στην Εικόνα 3.5, φαίνεται το παράθυρο που εμφανίζεται μετά την εκτέλεση του παραπάνω κώδικα.



Εικόνα 3.5 – Γραφική αναπαράσταση των κυττάρων στο Παιχνίδι της ζωής με OpenGL

ΚΕΦΑΛΑΙΟ 4

4 OpenCL

4.1 Εισαγωγή

Η τεχνολογία **OpenCL** (OpenComputingLanguage) είναι ένα ανοιχτό πρότυπο για την υλοποίηση εφαρμογών γενικού σκοπού, οι οποίες εκτελούνται σε ετερογενής πλατφόρμες αποτελούμενες από CPU, GPU και άλλους επεξεργαστές, δίνοντας έτσι την δυνατότητα στους προγραμματιστές να εκμεταλλευτούν στο έπακρο όλη την υπολογιστική ισχύ ενός συστήματος.

Αρχικά, το OpenCL κατασκευάστηκε από την Apple Inc, η οποία κατέχει και τα δικαιώματα του εμπορικού σήματος. Έπειτα, προτάθηκε το καλοκαίρι του 2008 στην ομάδα Khronos, όπου με τη συνεργασία πολλών εταιριών, όπως της AMD, IBM, Intel και Nvidia, το πρότυπο αυτό συνεχώς εξελίσσεται.

Το OpenCL περιλαμβάνει δύο βασικά στοιχεία:

1. Το API που χρησιμοποιείται για τον συντονισμό των παράλληλων εφαρμογών σε ετερογενής επεξεργαστές.
2. Μια γλώσσα προγραμματισμού ανεξαρτήτου πλατφόρμας, την OpenCL C (βασιζόμενη στην C99) .

4.2 Ορολογία

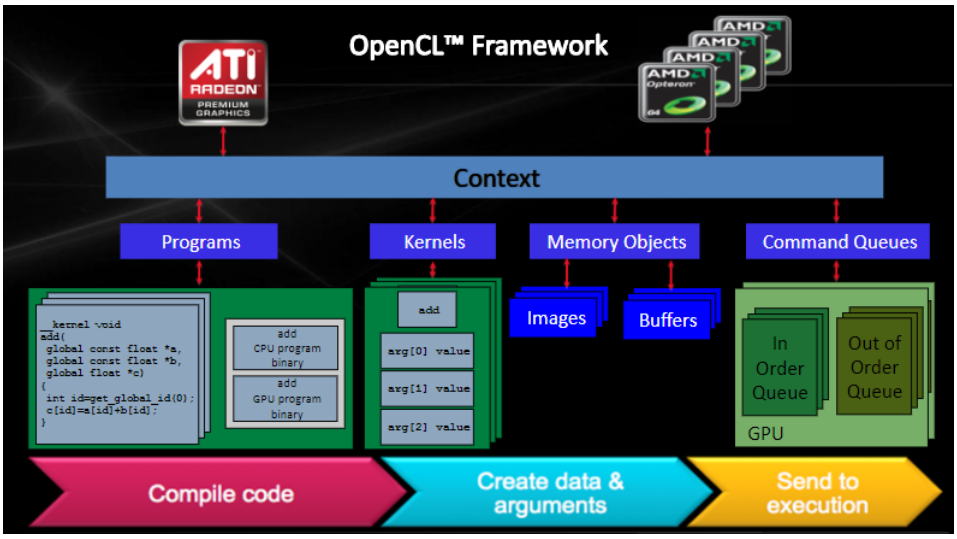
Μία εφαρμογή OpenCL αποτελείται από δύο μέρη. Το πρόγραμμα που εκτελείται στο **host (CPU)** και το πρόγραμμα που εκτελείται στις **συσκευές-device** (GPU). Παρακάτω δίνονται κάποιοι βασικοί όροι οι οποίοι πρέπει να είναι κατανοητοί πριν την κατασκευή μίας εφαρμογής OpenCL.

4.2.1 Platform

Μια πλατφόρμα αποτελείται από το host και ένα σύνολο συσκευών, τα οποία τα διαχειρίζεται το πλαίσιο εργασίας (framework) του OpenCL και επιτρέπει στην εφαρμογή να μοιράζεται τους διαθέσιμους πόρους και να εκτελεί τις συναρτήσεις kernel στις συσκευές.

•Είναι τύπου *cl_platform_id.*

•Αρχικοποιείται με την συνάρτηση *clGetPlatformIDs().*



Εικόνα 4.1 - OpenCL framework (16)

4.2.2 Device

Ο όρος "συσκευή", αναφέρεται σε οποιαδήποτε μονάδα επεξεργασίας σε ένα υπολογιστικό σύστημα. Εντός αυτής, υπάρχουν μία ή περισσότερες μονάδες υπολογισμού (compute unit), οι οποίες είναι υπεύθυνες για τον χειρισμό και την εκτέλεση των πράξεων. Μία τέτοια μονάδα, είναι μονάδα υλικού, η οποία εκτελεί υπολογισμούς ερμηνεύοντας κάθε φορά ένα σύνολο εντολών.

Έτσι, μία συσκευή OpenCL σε ένα υπολογιστικό σύστημα, μπορεί να θεωρηθεί είτε μία GPU, είτε μία πολυπύρηνη CPU ή και άλλοι επεξεργαστές. Για παράδειγμα, η κάρτα γραφικών GeForce GTX 280 έχει 30 μονάδες υπολογισμού.

•Είναι τύπου cl_device_id.

•Αρχικοποιούνται με την συνάρτηση clGetDeviceIDs().

4.2.3 Kernel

Kernel ονομάζουμε μία συνάρτηση υλοποιημένη με γλώσσα προγραμματισμού OpenCL C, η οποία περιέχει ένα σύνολο από εντολές που προορίζονται για μεταγλώττιση και εκτέλεση σε μία συσκευή. Μία τέτοια συνάρτηση προσδιορίζεται από το ειδικό αναγνωριστικό __kernel, το οποίο τοποθετείται πριν από το όνομα της. Πρέπει να σημειωθεί ότι μία συσκευή μπορεί να εκτελέσει και άλλες συναρτήσεις εκτός από τις

συναρτήσεις kernel, με τη διαφορά ότι οι kernel είναι σημεία εισόδου για την εφαρμογή μας. Με άλλα λόγια, οι kernel είναι συναρτήσεις οι οποίες μπορούν να κληθούν από τον host. Παρόλο που ο kernel τίθεται προς εκτέλεση από το πρόγραμμα host, το οποίο μπορεί να είναι υλοποιημένο σε C, C++, ή και άλλες γλώσσες, θα πρέπει να μεταγλωττιστεί ξεχωριστά, ώστε να παραμετροποιηθεί κατάλληλα για τη συσκευή στην οποία θα εκτελεστεί. Ο πηγαίος κώδικας του kernel μπορεί να είναι είτε σε ξεχωριστό αρχείο από αυτό του προγράμματος host (επέκτασης *.cl) είτε να βρίσκεται στο ίδιο αρχείο. Τέλος, η μεταγλώττιση του μπορεί να γίνει είτε κατά την εκτέλεση της host εφαρμογής, είτε να είναι ήδη μεταγλωττισμένη σε μορφή δυαδικού αρχείου. Για τη δημιουργία ενός αντικειμένου kernel:

•Είναι τύπου cl_kernel .

•Αρχικοποιείται με την συνάρτηση clCreateKernel().

4.2.4 Context

Ένα context δημιουργείται με μία ή περισσότερες συσκευές και χρησιμοποιείται για την διαχείριση αντικειμένων του OpenCL, όπως είναι το command queue (ουρές εντολών), τα αντικείμενα μνήμης (buffers) και οι συναρτήσεις kernel για την εκτέλεση στις διάφορες συσκευές.

•Είναι τύπου cl_context.

•Αρχικοποιείται με την συνάρτηση clCreateContext().

4.2.5 Command Queue

Μία ουρά εντολών, είναι ένα αντικείμενο το οποίο διαχειρίζεται εντολές που προορίζονται για εκτέλεση πάνω στις συσκευές. Αυτές οι εντολές αφορούν διάφορες λειτουργίες πάνω σε αντικείμενα μνήμης, προγράμματος και kernel. Οι εντολές τοποθετούνται με σειρά μέσα σε μία ουρά, αλλά η εκτέλεση τους μπορεί να είναι είτε με σειρά είτε χωρίς.

•Είναι τύπου cl_command_queue .

•Αρχικοποιείται με την συνάρτηση clCreateCommandQueue() .

4.2.6 Memory Objects

Τα αντικείμενα μνήμης είναι ένας τρόπος χειρισμού μίας περιοχής του μοντέλου μνήμης του OpenCL. Χρησιμοποιούνται για να δεσμεύσουν χώρο μνήμης σε μία συσκευή, ώστε να αποθηκευτούν εκεί τα δεδομένα του προγράμματος:

•Είναι τύπου cl_mem .

•Αρχικοποιούνται με την συνάρτηση clCreateBuffer().

4.2.7 Program

Ένα πρόγραμμα περιλαμβάνει ένα σύνολο από OpenCL kernel και μπορεί να περιέχει βοηθητικές συναρτήσεις, καθώς επίσης και δεδομένα σταθερών τα οποία συνεργάζονται με τις συναρτήσεις kernel. Για τη δημιουργία ενός αντικειμένου προγράμματος:

•Είναι τύπου cl_program .

•Αρχικοποιείται με την συνάρτηση clCreateProgramWithSource().

4.3 Το Παιχνίδι της Ζωής με OpenCL

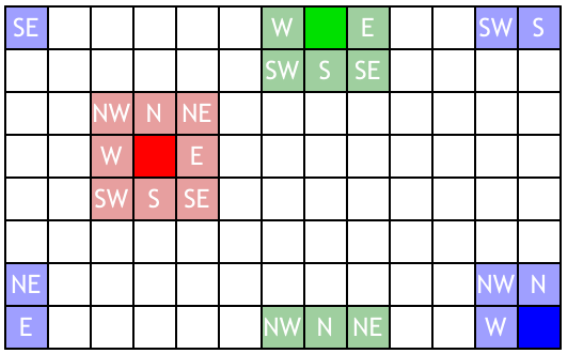
Παρακάτω παρουσιάζεται η υλοποίηση του Game of Life με υπολογισμό σε OpenCL ανεπτυγμένο από τον *David Pertiller* (17), όπου χρησιμοποιήθηκε το «OpenCL C++ Wrapper API» (18) με την προσθήκη της βιβλιοθήκης "cl.hpp".

4.3.1 Ορισμός του κυτταρικού κόσμου

Το παιχνίδι της ζωής είναι ένα κυτταρικό αυτόματο το οποίο προσομοιώνει την εξέλιξη της επιβίωσης ενός συνόλου κυττάρων σε ένα περιβάλλον. Το περιβάλλον της προσομοίωσης θεωρείται ένας «άπειρος» δισδιάστατος πίνακας, όπου σε κάθε κελί του αποθηκεύεται πληροφορία σχετικά με την κατάσταση του κυττάρου, ζωντανό ή νεκρό.
Σύμφωνα με το project, ο αρχικός κυτταρικός κόσμος βρίσκεται σε εξωτερικά αρχεία τα οποία περιέχουν με τυχαίο ρυθμό τους χαρακτήρες “**.**” και “x”. Τα αρχεία εισόδου που χρησιμοποιούνται αναπαριστούν το δισδιάστατο κυτταρικό κόσμο που οι τελείες “.” αναπαριστούν νεκρό κύτταρο, ενώ οι χαρακτήρες “x” αναπαριστούν ζωντανό κύτταρο. Το πλήθος των συνολικών κυττάρων είναι διαφορετικό σε κάθε αρχείο και κυμαίνεται από 1 εκατομμύριο έως 10 εκατομμύρια κύτταρα.

Ο κυτταρικός κόσμος στην πραγματική ζωή αποτελείται από άπειρα κύτταρα. Στους υπολογιστές, όμως, είναι αδύνατο να χρησιμοποιήσουμε άπειρη μνήμη. Για το

λόγο αυτό, χρησιμοποιούμε πίνακες που θεωρούνται κυκλικοί. Αυτό σημαίνει ότι ο αριστερός γείτονας της $1^{ης}$ στήλης θα είναι το κύτταρο που βρίσκεται στην τελευταία στήλη και αντίθετα. Αυτό παρουσιάζεται στην Εικόνα 4.2

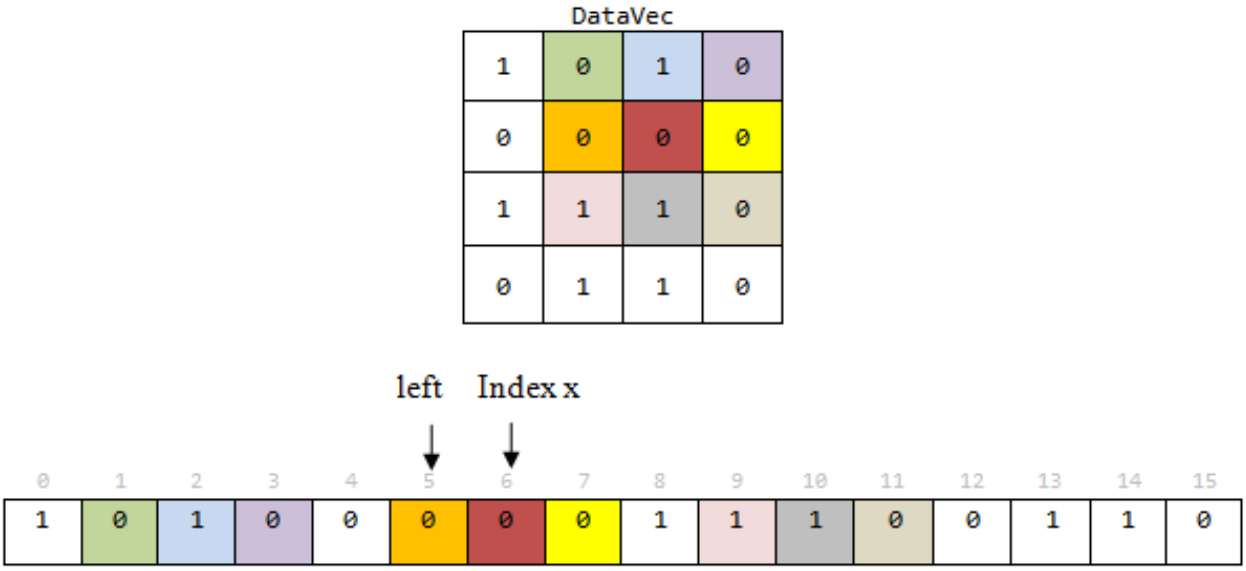


Εικόνα 4.2 – Αναπαράσταση του «άπειρου» κόσμου με κυκλικό πίνακα (19)

Από το αρχείο εισόδου και με κατάλληλο parser δημιουργείται ένας δισδιάστατος πίνακας, όπου περιέχει δύο πιθανά στοιχεία, το 0 και το 1. Όταν το κελί περιέχει το 1 θεωρείται ότι το κύτταρο είναι ζωντανό, σε αντίθετη περίπτωση θεωρείται νεκρό. Όμως, για μεγαλύτερη αποδοτικότητα του αλγορίθμου είναι προτιμότερη η μετατροπή του δισδιάστατου πίνακα σε ένα μονοδιάστατο, όπως φαίνεται στον κώδικα παρακάτω:

```
for (int y = 0; y < height; y++)
	for (int x = 0; x < width; x++)
		data[y*width + x] = dataVec[y][x];
```

Στην Εικόνα 4.3 ακολουθεί παράδειγμα όπου ένας πίνακα 4x4 δημιουργεί μονοδιάστατο πίνακα 16 θέσεων. Το στοιχείο στη θέση [1,2] με y=1 και x=2, αντιστοιχεί στην θέση y*4+x του δεύτερου πίνακα δηλαδή, στην θέση 6.



Εικόνα 4.3 – Παράδειγμα μετατροπής δισδιάστατου πίνακα σε μονοδιάστατο

Με την παραπάνω μετατροπή σε μονοδιάστατο πίνακα, ο κώδικας γίνεται λιγότερο ευανάγνωστος αλλά περισσότερο αποτελεσματικός για τον υπολογισμό της επόμενης κατάστασης του κάθε κυττάρου. Παρακάτω, αναλύεται το πρόγραμμα kernel καθώς και το προγραμμα host της συγκεκριμένης εφαρμογής με OpenCL.

4.3.2 Πρόγραμμα Kernel

Για τον υπολογισμό της επόμενης κατάστασης του κάθε κυττάρου χρησιμοποιείται η συνάρτηση Goll_All, η οποία θα φορτωθεί και θα μεταγλωττιστεί στη συσκευή OpenCL. Όταν δοθεί ο kernel για εκτέλεση, δημιουργείται ένα στιγμιότυπο για κάθε νήμα της εφαρμογής και εκτελείται ο ίδιος κώδικας αλλά σε διαφορετικά δεδομένα. Η διαφοροποίηση στα δεδομένα επιτυγχάνεται με την εντολή get_global_id(), όπου επιστρέφει το μοναδικό id του νήματος μέσα στο χώρο ευρετηριοποίησης. Τα ορίσματα που δέχεται η συνάρτηση είναι ο πίνακας data με τα δεδομένα, ο βοηθητικός πίνακας swapData για την καταγραφή των αποτελεσμάτων καθώς και οι αρχικές διαστάσεις του προβλήματος. Η συνάρτηση αθροίζει τους ζωντανούς γείτονες του κάθε κυττάρου,

υπολογίζει την επόμενη κατάσταση, με βάση τους κανόνες του παιχνιδιού και την αποθηκεύει στον βοηθητικό πίνακα swapData.

```
__kernel void Gol_All(
                          __global const int *data,
                          __global int *swapData,
                             int width,
                             int height
                         )
{
       // get index into global data array
       const int x = get_global_id(0);


       int top = x-width;
       int bottom = x+width;
       int left = -1; //in the inner ring, we can be sure that the index isn't out of
bounds without checking the borders
       int right = +1; //start position of the right neighbour

       if (x % width == 0) left += width;
       else if (x % width == (width - 1)) right -= width;
       if (top < 0) //row 0
              top += width*height;

       else if (bottom >= (height * width))
              bottom -= width*height;

       int alive = data[x+left]
                        + data[x+right]
                        + data[top+left]
                        + data[top]
                        + data[top+right]
                        + data[bottom+left]
                        + data[bottom]
                        + data[bottom+right];

//calculate next state according to the amount of alive cells in the neighbourhood
//a cell with 3 neigbours becomes/stays alive
//a live cell with 2 (or 3) neighbours stays alive (with 3 neighbours it already got
alive in the statement before)

       if ((alive == 3) ||   (alive == 2 && data[x] == 1)){
              swapData[x] = 1;
       }
       else{
              swapData[x] = 0;
       }
//dies from overpopulation or isolation, or remains dead if it already was

};
```

4.3.3 Πρόγραμμα Host

Το πρόγραμμα host ελέγχει και συντονίζει την εκτέλεση των kernel στις διάφορες συσκευές. Ο κώδικας που αφορά το OpenCL στο πρόγραμμα του host αποτελείται από τα παρακάτω:

- **Αρχικοποίηση του OpenCL**
  - Δημιουργία λίστας με τις διαθέσιμες πλατφόρμες.
  - Δημιουργία αντικειμένου τύπου Context.
  - Δημιουργία λίστας με τις διαθέσιμες συσκευές.
- **Φόρτωση του πηγαίου κώδικα του kernel και η μεταγλώττισή του**
- **Δημιουργία και φόρτωση των OpenCL buffers με τα απαραίτητα δεδομένα προς εκτέλεση στον kernel**
- **Εκτέλεση του kernel**
  - Ορισμός των παραμέτρων του kernel.
  - Δημιουργία ουράς εντολών command queue.
  - Εκτέλεση του kernel αντικειμένου.
- **Επιστροφή αποτελεσμάτων από την συσκευή.**

4.3.3.1 Αρχικοποίηση του OpenCL

Για τον συντονισμό και την εκτέλεση του kernel στη συσκευή απαιτείται να χρησιμοποιηθούν οι συναρτήσεις του API του OpenCL. Αρχικά, δημιουργείται ένα αντικείμενο πλατφόρμας OpenCL με την εντολή cl::Platform::get(&platform). Στη συνέχεια, δημιουργείται ένα context και αναζητούνται οι διαθέσιμες συσκευές που υπάρχουν στην πλατφόρμα.

```
// Get list of OpenCL platforms.
vector<cl::Platform> platform;
cl::Platform::get(&platform);

if (platform.empty()) {
	cerr << "OpenCL platforms not found." << endl;
	exit(-1);}

// Get first available GPU device which supports double precision.
cl::Context context;
vector<cl::Device> device;
for (auto p = platform.begin(); device.empty() && p != platform.end(); p++) {
	vector<cl::Device> pldev;

	try {
		p->getDevices(deviceType, &pldev);

		for (auto d = pldev.begin(); device.empty() && d != pldev.end(); d++) {
			if (!d->getInfo<CL_DEVICE_AVAILABLE>()) continue;

			device.push_back(*d);
			context = cl::Context(device);
```

```
                }
        }
        catch (...) {device.clear();}
}

if (device.empty()) {
        cerr << "No devices or no device with specified type found!" << endl;
        exit(-2);
}
```

4.3.3.2 Φόρτωση του πηγαίου κώδικα kernel και η μεταγλώττισή του

Ακολουθεί η φόρτωση του πηγαίου κώδικα kernel και η μεταγλώττισή του με τις εντολές cl::Program program(context, source) και program.build(device), όπως φαίνεται παρακάτω. Με την εντολή cl::Kernel gol(program, "Gol_All") δημιουργείται ένα αντικείμενο τύπου kernel από το ήδη μεταγλωττισμένο πρόγραμμα, όπου η συνάρτηση ονομάζεται «Gol_All».

```
std::string programString(stringifiedSourceCL);
cl::Program::Sources source(1, std::make_pair(programString.c_str(),
programString.length() + 1));

cl::Program program(context, source);
try
{
program.build(device);
}
catch (const cl::Error&)
{       cerr << "OpenCL compilation error" << endl <<
        program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(device[0]) << endl;
                        exit(-3);
}

cl::Kernel gol(program, "Gol_All");
```

4.3.3.3 Εκτέλεση του kernel αντικειμένου

Πριν την εκτέλεση, γίνεται η αρχικοποίηση των ορισμάτων για την συνάρτηση kernel. Δημιουργούνται οι buffers με τα κατάλληλα δεδομένα και με την εντολή gol.setArg() θέτονται τα ορίσματα στη συνάρτηση kernel. Στη συνέχεια, δημιουργείται η ουρά εντολών με την εντολή cl::CommandQueue queue() με παραμέτρους το context και τη συσκευή που δημιουργήθηκαν παραπάνω.

```
// Allocate device buffers and transfer input data to device.
```

```
cl::Buffer SwapBuffer(context, CL_MEM_WRITE_ONLY, datasize, NULL);

// Set kernel parameters.
cl_int clWidth = width;
cl_int clHeight = height;

gol.setArg(1, SwapBuffer);
gol.setArg(2, clWidth);
gol.setArg(3, clHeight);

// Create command queue.
cl::CommandQueue queue(context, device[0]);

// Launch kernel on the compute device.
cl::Buffer Buffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, datasize,
data.data());
gol.setArg(0, Buffer);

size_t global_item_size = ELEMENTS;
size_t local_item_size = 16; // Divide work items into groups of 64
```

Ακολουθεί η εκτέλεση του kernel στην συσκευή με την εντολή queue.enqueueNDRangeKernel(), η οποία θα εκτελεστεί επαναληπτικά με μια for, ώστε να υπολογιστεί η επόμενη κατάσταση κάθε κυττάρου για τις δεδομένες γενεές.

```
//till N-1 just copy output to input buffer (inexpensive way to read buffer during
iterations)
for (int gen = 0; gen < generations - 1; ++gen)
{
	queue.enqueueNDRangeKernel(gol, cl::NullRange, ELEMENTS, cl::NullRange);
	queue.enqueueCopyBuffer(SwapBuffer, Buffer, 0, 0, datasize);
}
```

4.3.3.4 Επιστροφή αποτελεσμάτων από τη συσκευή

Μετά τον υπολογισμό της τελικής κατάστασης των κυττάρων, προστίθεται μία ακόμα εντολή στην ουρά, η οποία λαμβάνει το αποτέλεσμα από την συσκευή και το αντιγράφει στην μνήμη του host. Αυτό επιτυγχάνεται με την κλήση της συνάρτησης queue.enqueueReadBuffer().

```
//in the last iteration, we read the buffer back to data
queue.enqueueNDRangeKernel(gol, cl::NullRange, ELEMENTS, cl::NullRange);
// Get result back to host.
queue.enqueueReadBuffer(SwapBuffer, CL_TRUE, 0, datasize, data.data());

}
```

ΚΕΦΑΛΑΙΟ 5

5 Συνεργασία OpenCL / OpenGL

5.1 Εισαγωγή

Το OpenCL και το OpenGL είναι δύο API που υποστηρίζουν αποτελεσματική λειτουργικότητα. Το OpenCL είναι ειδικά σχεδιασμένο για να αυξήσει την αποδοτικότητα των υπολογιστών σε όλες τις πλατφόρμες και το OpenGL είναι ένα δημοφιλές API γραφικών. Παρακάτω, θα αναλυθούν οι βασικές μέθοδοι για την κοινή χρήση πόρων και το συγχρονισμό μεταξύ αυτών των δύο API.

Το γενικό σενάριο συνεργασίας της OpenCL και OpenGL περιλαμβάνει την μεταφορά δεδομένων μεταξύ των δύο API με συχνό ρυθμό (πχ ανά frame) και κατεύθυνση αμφίδρομη.

Για παράδειγμα:

- Η OpenCL παράγει δεδομένα που αφορούν τις κορυφές σε μια προσομοίωση Φυσικής ώστε να τα αποδώσει γραφικά η OpenGL.
- Η OpenGL δημιουργεί ένα πλαίσιο εικόνας (frame) με περαιτέρω προ-επεξεργασία από την OpenCL.

Σε γενικές γραμμές, οι προγραμματιστές πρέπει συχνά να επιλέξουν ανάμεσα σε διαφορετικά API για το προγραμματισμό σε GPU, επιλέγοντας είτε πυρήνες σε GLSL, είτε σε OpenCL. Για σύντομες, απλές εργασίες που αλληλεπιδρούν άμεσα με τον αγωγό γραφικών, το OpenGL Compute Shaders μπορεί να είναι μια καλή επιλογή. Για πιο γενικά (και σύνθετα) σενάρια, ο υπολογισμός με OpenCL μπορεί να έχει πλεονεκτήματα σε σχέση με το GLSL, καθώς εκτελεί το τμήμα υπολογισμών ασύγχρονα στον graphics pipeline. Επίσης, το OpenCL επιτρέπει στην εφαρμογή να χρησιμοποιεί και άλλες συσκευές εκτός από μονάδες GPU. (20)

5.2 Η γενική ιδέα υλοποίησης

Για την επιτυχημένη υλοποίηση της συνεργασίας μεταξύ OpenCL και OpenGL απαιτείται η κατανόηση των παραμέτρων και των περιορισμών. Δεδομένου ότι οι περισσότερες κλήσεις OpenCL και OpenGL δεν εκτελούνται αμέσως αλλά τοποθετούνται σε ουρές εντολών (command queues), απαιτείται ένας συντονιστής για τον συντονισμό της δέσμευσης πόρων μεταξύ των δύο API.

Η αληθινή συνεργασία αφορά τη μεταβίβαση ιδιοκτησίας μεταξύ των δύο API και όχι των πραγματικών δεδομένων του πόρου. Με λίγα λόγια, τα δεδομένα δεσμεύονται κάθε φορά από ένα API, χωρίς να δημιουργείται αντίγραφό τους. Για να επιτευχθεί αυτό, ένα αντικείμενο μνήμης OpenCL δημιουργείται πάντα από ένα αντικείμενο OpenGL. Για παράδειγμα:

- Ένα texture στην OpenGL μετατρέπεται εύκολα σε εικόνα στην OpenCL με την εντολή *clCreateFromGLTexture*. Οι εικόνες στην OpenCL έχουν παρόμοια υλοποίηση με τα texture της OpenGL όπως είναι οι επιλογές των περιγραμμάτων, οι κανονικοποιημένες συντεταγμένες κ.α.
- Τα OpenGL (vertex) buffers μετατρέπονται σε OpenCL buffers με την εντολή *clCreateFromGLBuffer*.

Η συνεργασία της OpenGL και OpenCL στηρίζεται στο παρακάτω σενάριο:

1. Ένα αντικείμενο μνήμης της OpenCL δημιουργείται από παρόμοιο αντικείμενο μνήμης της OpenGL
2. Για κάθε πλαίσιο εικόνας (frame) , η OpenCL δεσμεύει το αντικείμενο μνήμης της, στη συνέχεια ενημερώνεται από την συνάρτηση της OpenCL (kernel) και τελικά απελευθερώνεται για να παρέχει τα ενημερωμένα δεδομένα πίσω στην OpenGL.
3. Για κάθε πλαίσιο εικόνας, η OpenGL αναπαριστά τα αποτελέσματα από το ενημερωμένο αντικείμενο μνήμης.

Υπάρχουν τρεις διαφορετικοί τρόποι με τους οποίους μπορεί να γίνει το σενάριο αυτό.

- Άμεση κοινή χρήση του OpenGL texture από την OpenCL με την εντολή clCreateFromGLTexture. Αποτελεί τον πιο αποδοτικό τρόπο συνεργασίας των

δύο API για συσκευή Intel HD Graphics και επιτρέπει την άμεση ανταλλαγή του texture.

- Δημιουργία ενός ενδιάμεσου Pixel-Buffer-Object (PBO) για το texture της OpenGL μέσω ***clCreateFromGLBuffer***. Στη συνέχεια, ακολουθεί η ενημέρωση του buffer με OpenCL και αντιγραφή των αποτελεσμάτων πίσω στο texture. Το μειονέκτημα αυτής της προσέγγισης είναι ότι παρόλο που η δημιουργία του PBO δεν αποτελεί νέο αντίγραφο, η τελική μεταφορά PBO σε texture εξακολουθεί να βασίζεται στην αντιγραφή, επομένως αυτή η μέθοδος είναι πιο αργή από τη μέθοδο άμεσης χρήσης που παρουσιάστηκε παραπάνω. Στη μέθοδο αυτή μπορούν να χρησιμοποιηθούν και τα Vertex-Buffer-Objects (VBO), όπως και τα Pixel-Buffer-Object (PBO) με την εντολή *clCreateFromGLBuffer.*
- Δημιουργία ενός δείκτη που δείχνει στο buffer της OpenGL μέσω της εντολής *glMapBuffer*. Η OpenCL επεξεργάζεται τον buffer μέσω του δείκτη και τέλος αντιγράφει τα αποτελέσματα πίσω με την εντολή *glUnmapBuffer*. Από πλευρά απόδοσης, θεωρείται πιο αργή ακόμα και από τη μέθοδο με τα PBO ειδικά σε μικρά texture, επειδή το buffer που δεσμεύει η OpenCL δημιουργείται / απελευθερώνεται σε κάθε πλαίσιο.

Η συνεργασία της OpenCL/OpenGL υλοποιείται στην OpenCL ως προέκταση (extension) που δημιούργησε η ομάδα Khronos. Το όνομα αυτής της επέκτασης είναι *cl_khr_gl_sharing* και πρέπει να υπάρχει μεταξύ των υποστηριζόμενων επεκτάσεων που διαθέτει η πλατφόρμα και η συσκευή για την ομαλή συνεργασία των OpenCL και OpenGL. Παρακάτω, παρουσιάζονται οι εντολές που χρησιμοποιούνται για την συνεργασία των δυο τεχνολογιών:

Πίνακας 5.1 – Εντολές στη συνεργασία OpenCL – OpenGL

Λειτουργία	**Περιγραφή**
clGetGLContextInfoKHR	Επιστρέφει τις συσκευές που σχετίζονται με το περιβάλλον της OpenGL
clCreateFromGLBuffer	Δημιουργεί έναν buffer OpenCL από έναν buffer της OpenGL
clCreateFromGLTexture	Δημιουργεί μια εικόνα (image) OpenCL από ένα texture της OpenGL
clCreateFromGLRenderbuffer	Δημιουργεί μια εικόνα 2D (image) OpenCL από το buffer εμφάνισης της OpenGL
clGetGLObjectInfo	Τύπος ερωτήματος που χρησιμοποιείται για τη δημιουργία του αντικειμένου μνήμης OpenCL. Επιστρέφει το τύπο και το όνομα του αντικείμενου της OpenGL που χρησιμοποιείται.
clGetGLTextureInfo	Παρέχει πρόσθετες πληροφορίες σχετικά με το αντικείμενο texture της OpenGL.
clEnqueueAcquireGLObjects	Η OpenCL αποκτά - δεσμεύει αντικείμενα μνήμης από την OpenGL
clEnqueueReleaseGLObjects	Η OpenCL απελευθερώνει – αποδεσμεύει αντικείμενα μνήμης που δέσμευσε από την OpenGL.

Δεδομένου ότι τα αντικείμενα μνήμης της OpenCL δημιουργούνται από τα αντίστοιχα αντικείμενα της OpenGL, χρειαζόμαστε κάποιο είδος κοινόχρηστου context ώστε να είναι συμβατό και με τις δύο τεχνολογίες. Μόλις δημιουργηθεί ένα κοινό context μεταξύ

OpenCL-OpenGL, η κοινή χρήση υλοποιείται χρησιμοποιώντας έναν από τους τρεις βασικούς τρόπους:

- Μέθοδος 1: Κοινή χρήση texture μέσω της εντολής *clCreateFromGLTexture*
- Μέθοδος 2: Κοινή χρήση texture μέσω αντικειμένων buffer-pixel (PBO) μέσω της εντολής *clCreateFromGLBuffer*
- Μέθοδος 3: Κοινή χρήση texture μέσω της εντολής *glMapBuffer*

Στην παρούσα διπλωματική εργασία, η συνεργασία των τεχνολογιών υλοποιείται με κοινή χρήση texture μέσω αντικειμένων vertex buffer (VBO).

5.3 Συγχρονισμός και ακεραιότητα δεδομένων

Κατά τον διαμοιρασμό των κοινόχρηστων αντικειμένων είναι σημαντικό να διασφαλισθεί η ακεραιότητα των δεδομένων. Συγκεκριμένα, πρέπει να διασφαλιστεί ότι η OpenGL έχει ολοκληρώσει όλες τις εκκρεμείς λειτουργίες που έχουν πρόσβαση στα κοινόχρηστα αντικείμενα , πριν η OpenCL δεσμεύσει τα αντικείμενα αυτά με την εντολή *clEnqueueAcquireGLObjects.* Αυτό διασφαλίζεται με την εντολή *glFinish*. Ομοίως, πρέπει να διασφαλιστεί ότι η OpenCL έχει ολοκληρώσει όλες τις εκκρεμείς λειτουργίες που έχουν πρόσβαση στα κοινόχρηστα αντικείμενα, πριν αποδεσμεύσει τα αντικείμενα. Αυτό διασφαλίζεται είτε με την κλήση της εντολής *clWaitForEvents*, είτε απλά καλώντας την εντολή *clFinish.* (20)

ΚΕΦΑΛΑΙΟ 6

6 Υλοποίηση OpenCL/OpenGL στο παιχνίδι της ζωής

6.1 Εισαγωγή

Στο κεφάλαιο αυτό θα αναλυθεί η συνεργασία των τεχνολογιών OpenCL / OpenGL στο παιχνίδι της ζωής. Από την φύση του προβλήματος, το παιχνίδι της ζωής ενδείκνυται για παράλληλη επεξεργασία καθώς ο υπολογισμός της επόμενης κατάστασης του κάθε κυττάρου πρέπει να υπολογιστεί ταυτόχρονα σε όλα τα κύτταρα. Όπως αναφέρθηκε σε προηγούμενο κεφάλαιο, η OpenCL και η OpenGL είναι δύο API που υποστηρίζουν αποτελεσματική συνεργασία. Το OpenCL είναι ειδικά σχεδιασμένο για να αυξήσει την αποδοτικότητα των υπολογιστών στην επεξεργασία και το OpenGL είναι ένα δημοφιλές API γραφικών. Παρακάτω, θα αναλυθεί η συνεργασία των δυο τεχνολογιών στο παιχνίδι της ζωής περιγράφοντας τα βήματα για την υλοποίηση του αλγορίθμου. Συγκεκριμένα, με βάση τον κώδικα που αναλύθηκε στο κεφάλαιο 4.3 (Το παιχνίδι της ζωής με OpenCL), θα αναφερθούν οι αλλαγές και οι προσθήκες ώστε να επιτευχθεί η συνεργασία των δύο τεχνολογιών και να αναπαραστήσουμε γραφικά το παιχνίδι της ζωής με την τεχνολογία OpenGL .

6.2 Γραφική αναπαράσταση του παιχνιδιού

Για την αναπαράσταση του παιχνιδιού με την OpenGL είναι απαραίτητη η δημιουργία του κεντρικού παραθύρου προβολής. Ο κώδικας υλοποίησης φαίνεται παρακάτω. Με την εντολή glutInit() ενεργοποιείται η βιβλιοθήκη GLUT η οποία είναι απαραίτητη για την εμφάνιση του παραθύρου. Στη συνέχεια, ορίζουμε το μέγεθος, την θέση, τον τύπο των γραφικών και το χρωματικό μοντέλο του παραθύρου με τις εντολές glutInitWindowSize(), glutInitWindowPosition(), glutInitDisplayMode() αντίστοιχα. Στην εφαρμογή χρησιμοποιείται παράθυρο με μεγέθος 1000x1000 και τεχνική της διπλής ενταμίευσης (double buffering), χρησιμοποιούνται δηλαδή δύο buffer χρωματικών τιμών ώστε να αποφευχθεί το τρέμουλο της οθόνης. Με την εντολή glutSetOption() καθορίζουμε ότι στο κλείσιμο του παραθύρου το πρόγραμμα θα συνεχίσει να εκτελείται.

```
	//Game of Life Window
	char *myargv[1];
	int myargc = 0;
	myargv[0] = strdup("My Game of Life");
	glutInit(&myargc, myargv);
	glutInitWindowSize(1000,1000);
	glutInitWindowPosition(20, 20);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
	glutCreateWindow("myLife");
	glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE,GLUT_ACTION_CONTINUE_EXECUTION);
	GLenum err = glewInit();
	if (GLEW_OK != err) {
		std::cout << "Failed to initialize" << std::endl;return(-1); }
	glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0, 1000, 0, 1000);
```

Έπειτα από τις ρυθμίσεις των παραμέτρων του παραθύρου, καλείται η glewInit() η οποία αρχικοποιεί την βοηθητική βιβλιοθήκη GLEW (OpenGL Extension Wrangler). Η GLEW απλοποιεί τις λειτουργίες της OpenGL και καθιστά την εφαρμογή περισσότερο ευέλικτη για τον προγραμματιστή. Με την εντολή glClearColor() αρχικοποιείται ο χρωματικός buffer του υπολογιστή και τίθεται το φόντο με μαύρο χρώμα. Με την εντολή glMatrixMode(GL_PROJECTION) καθορίζεται ότι θα τροποποιηθεί το μητρώο προβολής και σε συνδυασμό με την εντολή gluOrtho2D καθορίζεται η απεικόνιση της σκηνής στο επίπεδο δύο διαστάσεων και ορίζονται οι συντεταγμένες αποκοπής στο μέγεθος του παραθύρου.

Μετά την αρχικοποίηση του παραθύρου, έπεται να ορισθούν οι βασικοί buffers για την απεικόνιση του κυτταρικού κόσμου. Η OpenGL αναπαριστά σχήματα ορίζοντας τις κορυφές και τον χρωματισμό τους ανεξάρτητα. Στην εφαρμογή αυτή, για την αποθήκευση αυτών των στοιχείων χρησιμοποιούνται Array Buffer Objects.

6.3 Δημιουργία και αρχικοποίηση των Vertex Buffer Objects (VBO)

Για την γραφική απεικόνιση των κυττάρων στην οθόνη, η OpenGL χρειάζεται τις συντεταγμένες όλων των κυττάρων που πρόκειται να εμφανιστούν καθώς και το χρώμα τους. Τα Buffer Objects αποθηκεύουν αυτές τις πληροφορίες ως εξής: Κάθε κύτταρο ισοδυναμεί με ένα σημείο στην οθόνη και διαθέτει :

- Δυο συντεταγμένες x,y (απεικόνιση σε 2D χώρο)

- ένα χρώμα που αναπαριστάται στην μορφή RGB (r,g,b) και η τιμή του εξαρτάται από την κατάσταση του κυττάρου.

Η δημιουργία των buffers στην OpenGL υλοποιείται με την βοήθεια των εντολών glGenBuffers(), glBindBuffer() και glBufferData(). Τα αντικείμενα στην OpenGL χαρακτηρίζονται από έναν ακέραιο μοναδικό αριθμό που λειτουργεί ως "λαβή" (handle). Ο προγραμματιστής μπορεί να ορίσει αυθαίρετα έναν αριθμό αλλά συνήθως προτείνεται να αναλάβει η OpenGL την παραγωγή αυτού του αριθμού με την εντολή glGenBuffers. Με την εντολή αυτή δεν δεσμεύει μνήμη αλλά απλά δεσμεύεται ο συγκεκριμένος αριθμός-λαβή για το buffer. Για να δεσμευτεί μνήμη για το buffer χρειάζεται να αντιστοιχηθεί αυτή η «λαβή» με το context της OpenGL με την εντολή glBindBuffer(GL_ARRAY_BUFFER, hPobj). Στη συνέχεια, δεσμεύεται μνήμη για το buffer με την εντολή glBufferData(), όπου δηλώνεται το μέγεθος του πίνακα. Για τον πίνακα με τις συντεταγμένες των σημείων, είναι απαραίτητη η αποθήκευση δύο ακεραίων που αντιστοιχούν στα x και y κάθε σημείου. Μετά την δέσμευση μνήμης, ακολουθεί η προσθήκη δεδομένων στον πίνακα. Με την εντολή glMapBuffer αντιστοιχίζεται ο πίνακας της GPU (hPobj) με ένα πίνακα της εφαρμογής (points). Με την εντολή αυτή, συμπληρώνεται ο array buffer hPobj μέσω του πίνακα points. Στο τέλος, αποδεσμεύεται ο πίνακας με την εντολή glUnmapBuffer(). Παρακάτω, φαίνεται σε μικρογραφία η αποθήκευση των συντεταγμένων στον πίνακα points για ένα πίνακα 3x4.(βλ. Εικόνα 6.1)

```
//θέτουμε τον buffer hPobj με τις συντεταγμένες των σημείων για την openGL.
//Για  κάθε σημείο χρησιμοποιώ 2 int που ισοδυναμούν στα x,y
GLuint hPobj;
glGenBuffers(1, &hPobj);
glBindBuffer(GL_ARRAY_BUFFER, hPobj);
glBufferData(GL_ARRAY_BUFFER, 2 * sizeof(int) * ELEMENTS, NULL,GL_DYNAMIC_DRAW);
points =(int *) glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);
m = 0;
for (int i = 0; i < height; i++)
{
	for (int j = 0; j < width; j++)
	{
		points[m] = i;
		points[m + 1] = j;
		m = m + 2;
	}
}
glUnmapBuffer(GL_ARRAY_BUFFER);
```

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

Points

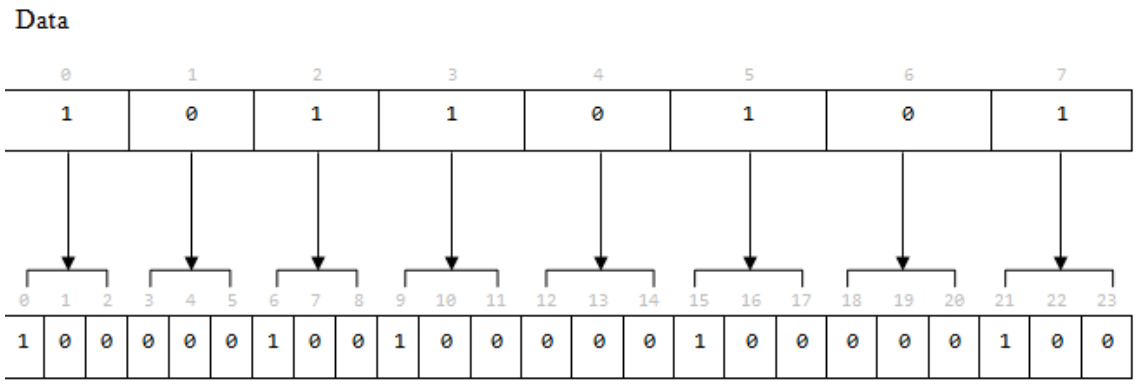
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
0	0	0	1	0	2	0	3	1	0	1	1	1	2	1	3	2	0	2	1	2	2	2	3

Εικόνα 6.1 – Μετατροπή δισδιάστατου πίνακα συντεταγμένων σε μονοδιάστατο

Με τον ίδιο τρόπο, δημιουργείται και αρχικοποιείται ο πίνακας που αποθηκεύονται τα χρώματα των κυττάρων. Στο buffer των χρωμάτων αποθηκεύεται για κάθε κύτταρο η ακολουθία 3 αριθμών που αντιστοιχούν στα διαθέσιμα βασικά χρώματα r,g,b (red,green,blue). Για την αναπαράσταση των ζωντανών κυττάρων το χρώμα ορίζεται ως κόκκινο, ενώ για τα υπόλοιπα κύτταρα το χρώμα ορίζεται μαύρο. Η συμπλήρωση του πίνακα εξαρτάται από τις τιμές που περιέχει ο αρχικός πίνακας data της εφαρμογής, ο οποίος περιέχει την τιμή 1 (ζωντανό κύτταρο) ή την τιμή 0 (νεκρό κύτταρο).

```
//προετοιμασία των buffers για την OpenGL
//ο buffer hCobj περιέχει σε κάθε κελί του 3 float με τα διαθέσιμα χρώματα r,g,b
//δεσμεύουμε χωρο
glGenBuffers(1, &hCobj);
glBindBuffer(GL_ARRAY_BUFFER, hCobj);
glBufferData(GL_ARRAY_BUFFER, 3 * sizeof(float)*ELEMENTS, NULL,GL_DYNAMIC_DRAW);
colors = (float*) glMapBuffer(GL_ARRAY_BUFFER, GL_READ_WRITE);

//Γέμισμα του πινάκα colors συμφώνα με το πινάκα data.
//Χρησιμοποιούμε μόνο το κόκκινο.
int m = 0;
for (int i = 0; i <ELEMENTS; i ++) {
	if (data[i] == 1) {
		colors[m] = 1;
	}
	else {
		colors[m] = 0;
	}
	colors[m+1] = 0;
	colors[m+2] = 0;
	m = m + 3;
```

```
        }
        glUnmapBuffer(GL_ARRAY_BUFFER);
        glBindBuffer(GL_ARRAY_BUFFER,0);
```



Εικόνα 6.2 – Μετατροπή κάθε στοιχείο του πίνακα data σε ακολουθία 3 χρωμάτων (r,g,b)

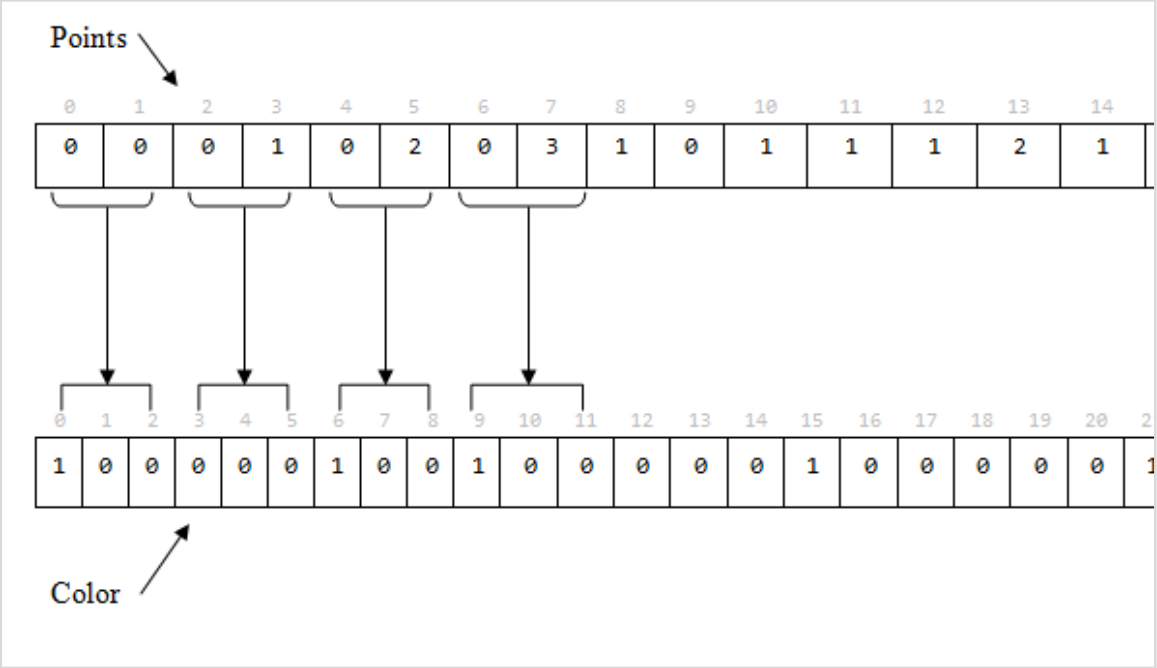
6.4 Προβολή των Buffers στην οθόνη

Η OpenGL χρειάζεται τον buffer με τις συντεταγμένες κάθε κυττάρου (hPobj) και τον buffer με το χρώμα κάθε κυττάρου (hCobj). Με την εντολή glVertexPointer(2, GL_INT, 0, 0) διευκρινίζεται στην OpenGL ότι για την αναπαράσταση των δεδομένων χρησιμοποιούνται 2 τιμές για τις συντεταγμένες οι οποίες είναι τύπου GL_INT. Ομοίως για τα χρώματα, με την εντολή glColorPointer(3, GL_FLOAT, 0, 0) διευκρινίζεται στην OpenGL ότι για τον χρωματισμό χρησιμοποιούνται 3 τιμές τύπου GL_FLOAT. Επίσης, ενεργοποιείται η χρήση μητρώων σημείων και χρωμάτων με την εντολή glEnableClientState(). Παρακάτω, φαίνεται η συνάρτηση display() που καλείται επαναληπτικά στο κύκλο της εντολής glutMainLoop(), όπως παρουσιάστηκε στο κεφάλαιο 3.6.9.

```
        void display()
        {
                glBindBuffer(GL_ARRAY_BUFFER, hPobj);
                glVertexPointer(2, GL_INT, 0, 0);
                glEnableClientState(GL_VERTEX_ARRAY);

                glBindBuffer(GL_ARRAY_BUFFER, hCobj);
                glColorPointer(3, GL_FLOAT, 0, 0);
                glEnableClientState(GL_COLOR_ARRAY);
```

```
            glPointSize(0.5);
            glDrawArrays(GL_POINTS, 0,wglobal*hglobal);

            glDisableClientState(GL_VERTEX_ARRAY);
            glDisableClientState(GL_COLOR_ARRAY);
            glBindBuffer(GL_ARRAY_BUFFER,0);

            glutSwapBuffers();
            glFlush();
            return;
      }
```

Συνοψίζοντας, κάθε σημείο (κύτταρο) αναπαρίσταται στην OpenGL από 2 τιμές συντεταγμένων (x,y) και 3 τιμές χρώματος (r,g,b). Στην Εικόνα 6.3 παρουσιάζεται η συσχέτιση των πινάκων για κάθε κύτταρο.



Εικόνα 6.3 – Γραφική αναπαράσταση των Buffer της OpenGL.

6.5 Αρχικοποίηση της OpenCL

Για την συνεργασία με την OpenGL απαραίτητη προϋπόθεση είναι η δημιουργία κοινού context και η ύπαρξη του extension "**cl_khr_gl_sharing**", όπως αναλύθηκε στο κεφάλαιο 5 (σελ 36). Ο κώδικας για την αρχικοποίηση της OpenCL παραμένει ίδιος, όπως αναλύθηκε στο κεφάλαιο 4.3.3.1 (βλ. σελ. 33). Η διαφορά βρίσκεται στη δημιουργία κοινού context OpenCL/OpenGL, το οποίο επιτυγχάνεται με την μέθοδο contextFromTypeGL(), που δημιουργήσαμε για την συνεργασία των τεχνολογιών.

```
vector<cl::Platform> platform; // Get list of OpenCL platforms.
cl::Platform::get(&platform);
if (platformId > -1 && platformId < platform.size())
{
       cl::Platform userSpecifiedPlatform = platform[platformId];
       platform.clear();
       platform.push_back(userSpecifiedPlatform);
}

//Get first available GPU device which supports double precision.

//Get Context to share with OpenGL
cl::Context context = contextFromTypeGL(deviceType);

vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES >();
std::string programString(stringifiedSourceCL);
cl::Program::Sources source(1,
std::make_pair(programString.c_str(), programString.length() + 1));
cl::Program program(context, source);
try
{
program.build(devices);
}
catch (const cl::Error&)
{ cerr << "OpenCL compilation error" << endl <<
   program.getBuildInfo<CL_PROGRAM_BUILD_LOG>(devices[0]) << endl; exit(-3);
}

// Create command queue.
queue = cl::CommandQueue(context, devices[0]);
gol= cl::Kernel(program, "Gol_All");
```

Η μέθοδος contextFromTypeGL() βρίσκεται διαθέσιμη στο βιβλίο «Using OpenCL: Programming Massively Parallel Computers. J. Kowalik, T. Puźniakowski.» (21) και παρουσιάζεται παρακάτω:

```
//Δημιουργει και επιστρέφει ενα cl::context για την συνεργασία με την OpenGL
cl::Context contextFromTypeGL(cl_device_type devtype) {
       std::vector < cl::Device > devices;
```

```
std::vector < cl::Device > device;
std::vector < cl::Platform > platforms;
cl::Platform::get(&platforms);
for (size_t i = 0; i < platforms.size(); i++) {
try {
platforms[i].getDevices(devtype, &devices);
if (devices.size() > 0) {
       for (size_t j = 0; j < devices.size(); j++) {
       if (checkExtension(devices[j], "cl_khr_gl_sharing")) {
#if defined(linux)
cl_context_properties cps[] = {
CL_GL_CONTEXT_KHR, (cl_context_properties)glXGetCurrentContext(),
CL_GLX_DISPLAY_KHR, (cl_context_properties)glXGetCurrentDisplay(),
CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[i])(),
0
};
#else // Win32
cl_context_properties cps[] = {
CL_GL_CONTEXT_KHR, (cl_context_properties)wglGetCurrentContext(),
CL_WGL_HDC_KHR, (cl_context_properties)wglGetCurrentDC(),
CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[i])(),
0
};
#endif
device.push_back(devices[j]);
return cl::Context(device, cps, NULL, NULL);}}
}
}
catch (cl::Error e) { // this is thrown in case of 0 devices of specified type
                      std::cout << e.what();
}
}
throw cl::Error(-1, "No such device");
}
```

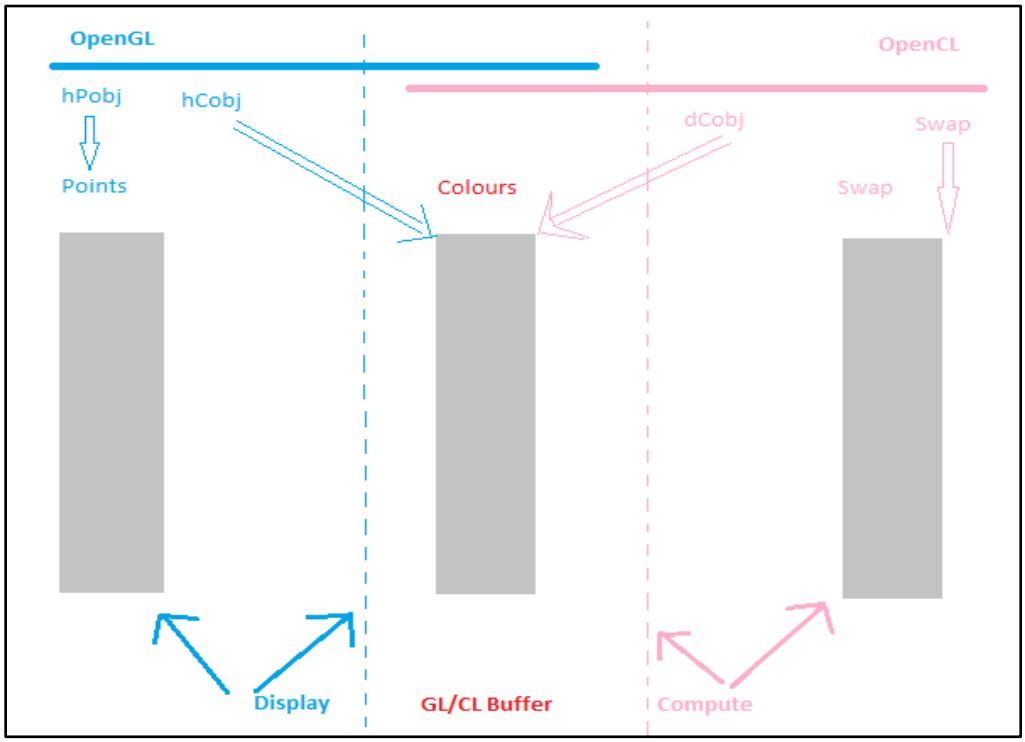
6.6 Δημιουργία των Buffers στην συσκευή

Για τον υπολογισμό της επόμενης κατάστασης κάθε κυττάρου είναι απαραίτητος ένας OpenCL Buffer. Ο OpenCL Buffer πρέπει να δημιουργηθεί από τον αντίστοιχο Buffer της OpenGL, όπως ορίζει ο η ομάδα Khronos στον οδηγό συνεργασίας OpenGL/OpenCL (“OpenCL memory object is created from the OpenGL texture.” (20)). Με την εντολή cl::BufferGL(context, CL_MEM_READ_WRITE, hCobj) δημιουργούμε το στιγμιότυπο της OpenCL για τον κοινό πίνακα με τον δείκτη dCobjCL. Ως παράμετρος ορίζεται το κοινό context με την OpenGL που δημιουργήσαμε προηγουμένως, η τιμή CL_MEM_READ_WRITE (flag που σημαίνει ότι ο kernel έχει δικαίωμα για εγγραφή και ανάγνωση) και τον OpenGL Buffer που αποθηκεύεται το χρώμα hCobj. Η OpenCL επεξεργάζεται τις τιμές του πίνακα ενώ η OpenGL τον αναπαριστά γραφικά. Επίσης, στη συνέχεια δημιουργούμε και τον πίνακα SwapBuffer

για να καταχωρεί η OpenCL τη νέα κατάσταση κάθε κυττάρου. Ο τελευταίος δεν απαιτείται για την γραφική αναπαράσταση.

```
        //θέτουμε τους buffers για την OpenCL: shared buffer dCobjCL και τον Swap
       //copy from hCobj
       dCobjCL = cl::BufferGL(context, CL_MEM_READ_WRITE, hCobj);

       SwapBuffer = cl::Buffer(context ,CL_MEM_WRITE_ONLY,
                                          3*ELEMENTS*sizeof(cl_int) , NULL);
```

Στην παρακάτω Εικόνα 6.4 φαίνεται ο διαμοιρασμός του buffer του χρώματος από τις δύο τεχνολογίες.



Εικόνα 6.4 - Διαμοιρασμός του Colour buffer από τις δύο τεχνολογίες OpenCL/OpenGL

6.7 Εισαγωγή παραμέτρων στον kernel

Με τις εντολές παρακάτω ορίζονται οι παράμετροι του αντικειμένου kernel.

```
       // Set kernel parameters.
       gol.setArg(0, dCobjCL);
       gol.setArg(1, SwapBuffer);
       gol.setArg(2, clWidth);
```

```
gol.setArg(3, clHeight);
```

Οι παράμετροι πρέπει να αντιστοιχίζονται μια προς μια με την λίστα των ορισμάτων στον kernel

```
__kernel void Gol_All(
                        __global float *data,
                       __global float *swap,
                       int width,
                       int height
                          )
```

Όπου data, swap είναι buffers για τον υπολογισμό του χρώματος και width, height το μέγεθος του κυτταρικού χώρου. Τα συνολικά κύτταρα που υπάρχουν είναι το γινόμενο width*height (ELEMENTS).

6.8 Υπολογισμός στον Kernel

Το χρώμα κάθε κυττάρου αναπαρίσταται στον buffer data με την μορφή (r,g,b). Παρόλα αυτά, η σημαντική πληροφορία βρίσκεται μόνο στο κανάλι (r) και αυτό γιατί:

- Αν το κύτταρο είναι ζωντανό έχει κόκκινο χρώμα, δηλ. 1,0,0.
- Αν το κύτταρο είναι νεκρό έχει μαύρο χρώμα, δηλ. 0,0,0.

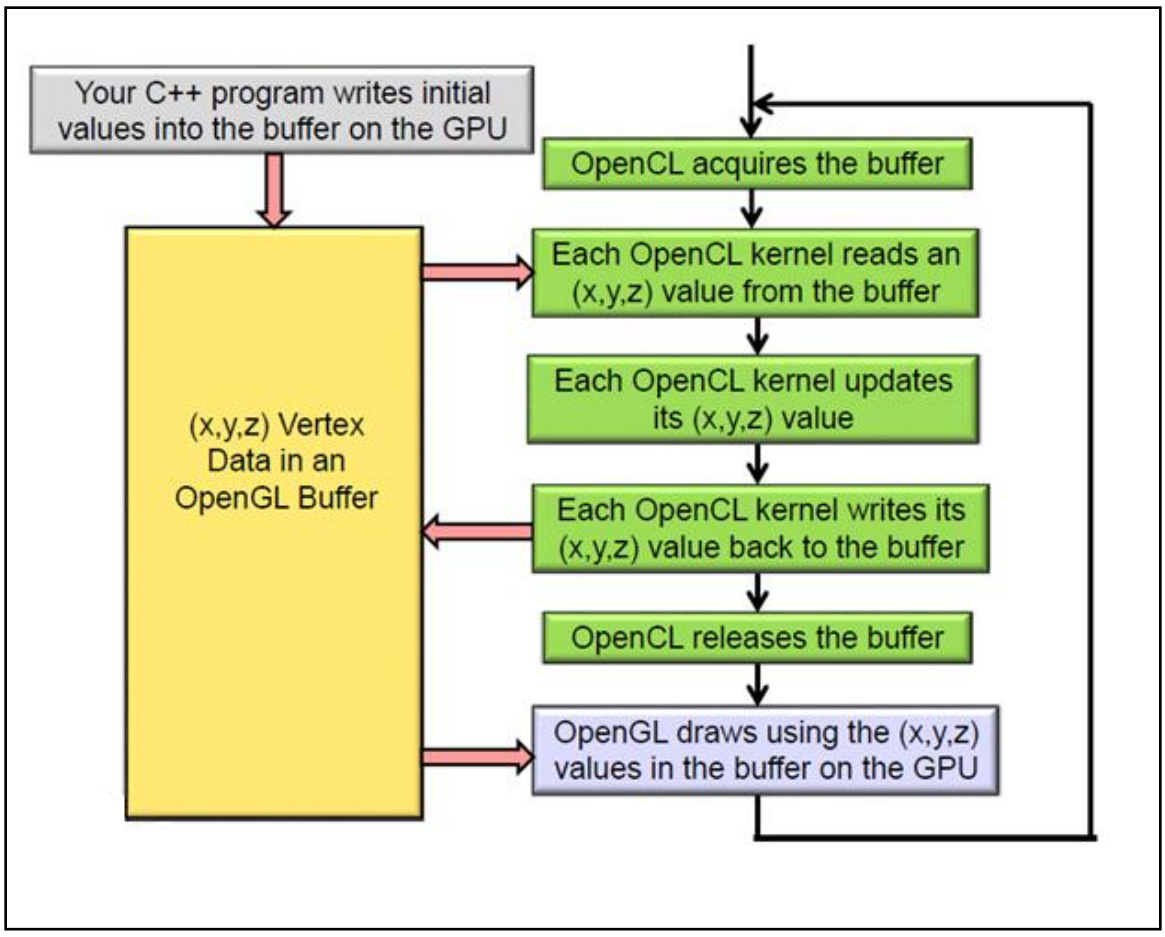
Έτσι λοιπόν, για κάθε κύτταρο αρκεί μόνο το R κανάλι, το οποίο βρίσκεται στον buffer σε θέσεις πολλαπλάσιες του 3 (δηλαδή στις θέσεις 0,3,6,9 κλπ). Συνεπώς, η συνάρτηση kernel πρέπει να υπολογίζει τη νέα κατάσταση των κυττάρων από τα index που αφορούν το κανάλι (r), δηλαδή να μόνο τις θέσεις πολλαπλάσιες του 3. Χρησιμοποιούμε το τελεστή modulo για τον διαχωρισμό των καναλιών. Αν το υπόλοιπο με τον αριθμό 3 είναι μηδέν τότε το νήμα υπολογίζει το κανάλι R, διαφορετικά θέτει στο αποτέλεσμα την τιμή 0.

```
// get index into global data array
const int x = get_global_id(0);

// index calculates the (R) channel
if (x%3==0)
{
       int top = x-width*3;
       int bottom = x+width*3;
       int left = -3  //start position of the right neighbor
       int right = +3;  //start position of the right neighbor
```

```
            If (x % (width*3)== 0)
                  left += width*3;
            else if (x % (width*3 ) == (width*3 - 3))
                  right -= width*3;

            If (top < 0) //row 0
                  top += width*height*3;

            else if (bottom >= (height * 3* width))
                  bottom -= 3*width*height;

            int alive = data[x+left]
                              + data[x+right]
                              + data[top+left]
                              + data[top]
                              + data[top+right]
                              + data[bottom+left]
                              + data[bottom]
                              + data[bottom+right];

            If ((alive == 3) ||   (alive == 2 && data[x] == 1)){
                  swap[x] = 1;
            else
                  swap[x] = 0;
      }
      else
            swap[x] =0;
```

6.9 Αναπαράσταση του κυτταρικού κόσμου με κίνηση

Η βασική ιδέα (όπως αναφέρθηκε στο κεφάλαιο 5.2) στηρίζεται στην εκμετάλλευση του κοινού buffer από τις τεχνολογίες OpenCL / OpenGL. Και οι δύο τεχνολογίες μπορούν να χρησιμοποιήσουν τον Buffer ανά πάσα στιγμή, αλλά όχι συγχρόνως. Η OpenCL οφείλει να δεσμεύσει τον Buffer από την OpenGL για να κάνει τους υπολογισμούς και τέλος να αποδεσμεύσει τον Buffer πίσω στην OpenGL ώστε η τελευταία να αναπαραστήσει τα αποτελέσματα στην οθόνη. Η παραπάνω διαδικασία γίνεται επαναληπτικά, με αποτέλεσμα να δίνεται η εντύπωση της κίνησης των κυττάρων. Η Εικόνα 6.5 δείχνει παραστατικά την συνεργασία των OpenCL/OpenGL με κοινό διαμοιραζόμενο Buffer (22).



Εικόνα 6.5 - OpenCL / OpenGL συνεργασία με Vertex Buffer Objects (VBO) (22)

Όπως φαίνεται στην εικόνα, η OpenCL υπολογίζει τη νέα κατάσταση του κυττάρου, όταν η OpenGL δεν χρησιμοποιεί τον Buffer και παραμένει στάσιμη. Για τον λόγο αυτό χρησιμοποιείται η “Idle Function” της OpenGL και μέσω αυτής η OpenCL κάνει τον υπολογισμό της. Η γραφική αναπαράσταση γίνεται με βασικές τις εντολές glutDisplayFunc(), glutIdleFunc() και glutMainLoop() όπου:

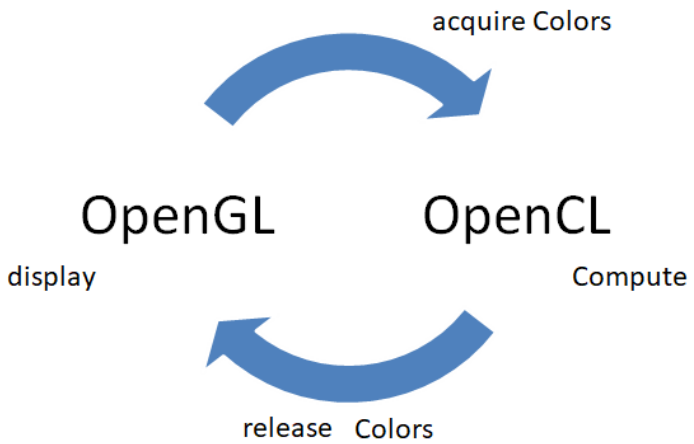
- glutDisplayFunc(): ενημερώνει την OpenGL τι να σχεδιάσει
- glutIdleFunc(): εκτελεί τον κώδικα όταν η OpenGL μένει στάσιμη
- glutMainLoop(): ενεργοποιεί τον κύκλο διαχείρισης γεγονότων (event processing loop)(βλ. κεφάλαιο 3.6.9).

```
glutDisplayFunc(display);
glutIdleFunc(animate);
glutMainLoop();
```

Η συνάρτηση animate() είναι παράμετρος της glutIdleFunc() και εκτελείται κάθε φορά που η OpenGL δεν εκτελεί εργασία. Παρακάτω, παρουσιάζεται ο κώδικας της συνάρτησης animate().

```
void animate()
{
	glFinish();

	//Η OpenCL δεσμεύει τον buffer
	acquireFromGLtoCL(dCobjCL);

	// Εκτελείται ο kernel στην συσκευή
	queue.enqueueNDRangeKernel(gol, cl::NullRange, 3 * ELEMENTS, cl::NullRange, 0);

	// Αντιγραφή των αποτελεσμάτων από τον Swap Buffer στον αρχικό.
	queue.enqueueCopyBuffer(SwapBuffer, dCobjCL, 0, 0,sizeof(cl_float)*3* ELEMENTS);

	// Η OpenCL αποδεσμεύει τον buffer
	releaseObject(dCobjCL);

	//Εμφάνιση αποτελεσμάτων στην οθόνη
	glutPostRedisplay();

	If (generations == max_generations) glutDestroyWindow(glutGetWindow());
 }
```

Με την τελευταία συνθήκη if, το παράθυρο προβολής κλείνει αυτόματα με την εντολή glutDestroyWindow() όταν ο αριθμός των γενεών ξεπεράσει το όριο που έχουμε θέσει στις παραμέτρους του προγράμματος.



Εικόνα 6.6 – Γραφική αναπαράσταση της συνάρτησης animate()

ΚΕΦΑΛΑΙΟ 7

7 Αποτελέσματα - πειράματα

7.1 Ορθότητα αποτελεσμάτων

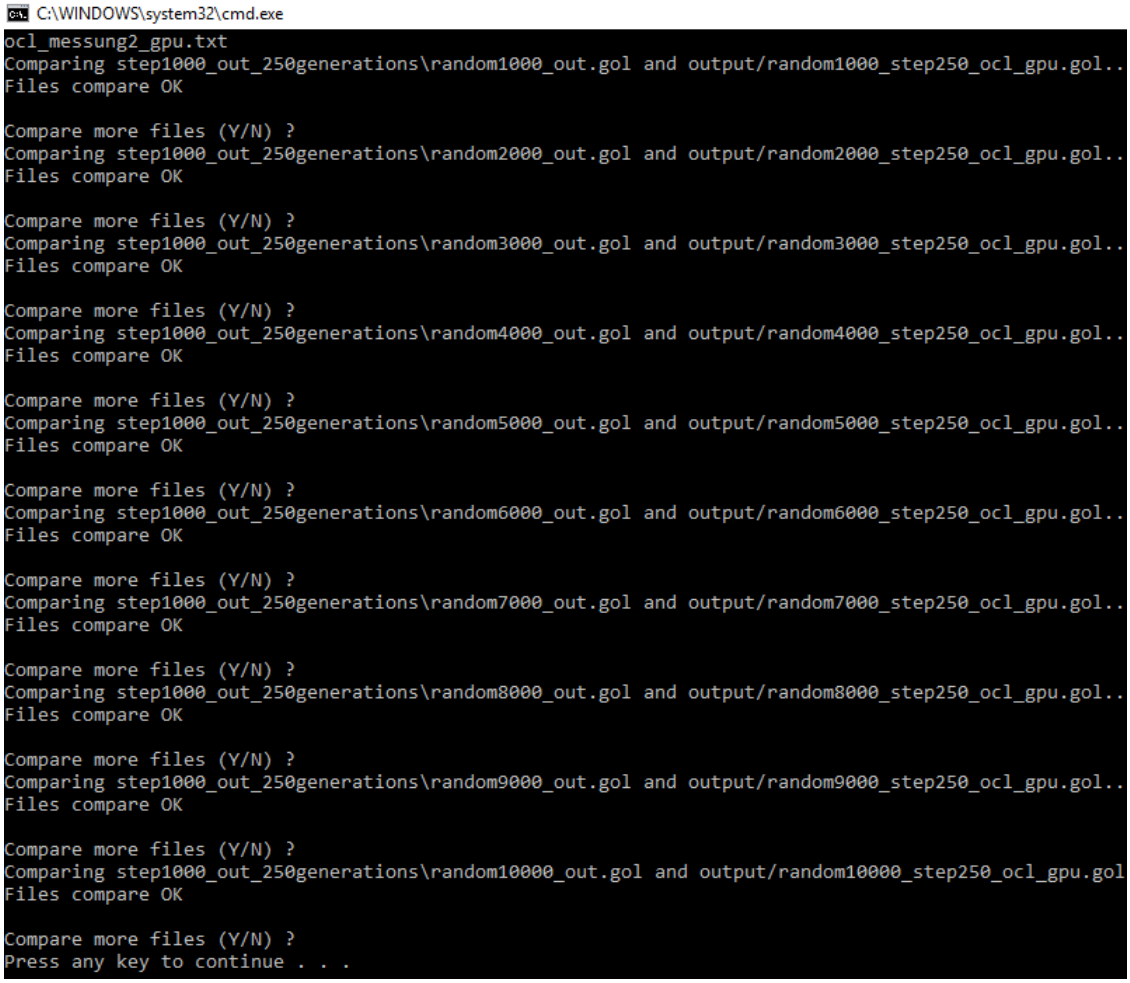
Στόχος της παρούσας εργασίας είναι η γραφική αναπαράσταση του κυτταρικού κόσμου με OpenGL και η συνεργασία της με την OpenCL. Η εξέλιξη της επιβίωσης του κάθε κυττάρου αναπαρίσταται με κινούμενα γραφικά για 250 γενεές. Για να εξασφαλίσουμε την ακεραιότητα και την ορθότητα των αποτελεσμάτων, χρησιμοποιήθηκαν τα αρχεία εισόδου από το αρχικό project (David Pertiller (17), (23)) καθώς και τα αρχεία με τα έγκυρα αποτελέσματα μετά από 250 γενεές. Έτσι, στο τέλος της εκτέλεσης του προγράμματος, αντιγράφονται τα αποτελέσματα από το SwapBuffer πίσω στην CPU με την εντολή enqueueReadBuffer(). Στη συνέχεια, αντιγράφουμε μόνο το χρωματικό κανάλι (R) σε νέο πίνακα και μεταφέρουμε τα αποτελέσματα σε 2Δ πίνακα ώστε να συγκριθούν με τα σωστά αποτελέσματα.

```
	vector<float> dataBack(3*ELEMENTS);
	vector<float> data2(ELEMENTS);
	queue.enqueueReadBuffer(SwapBuffer,CL_TRUE, 0, 3*(sizeof(cl_float)*ELEMENTS),
				dataBack.data());
	m = 0;
	for (int i = 0; i < 3*ELEMENTS; i++)
	{
		if (i % 3 == 0){
			data2[m] = dataBack[i]; //copy only the R channel to data2
			m = m + 1;
		}
	}

	//transfer results to 2d array
	for (int i = 0; i < height; i++)
	{
		for (int j = 0; j < width; j++)
		{
			dataVec2[i][j] = data2[i*width + j];
		}
	}
```

Για όλα τα αρχεία εισόδου που δόθηκαν, το πρόγραμμα παρήγαγε σωστούς υπολογισμούς. Το πρόγραμμα εκτελέστηκε με διαφορετικά μεγέθη εισόδου, ξεκινώντας από 1 εκ. και καταλήγοντας σταδιακά στα 10 εκ. κύτταρα. Για τον έλεγχο ορθότητας του

αποτελέσματος χρησιμοποιήθηκε η συνάρτηση CompareData που δημιουργήθηκε για τον σκοπό αυτό.

```
bool CompareData(vector <vector<int>> &boardData, vector <vector<int>> &mustHaveData) {
	unsigned int height = boardData.size();
	unsigned int width = boardData[0].size();
	bool same = true;
	for (unsigned int i = 0; i < height; i++)
	{
		for (unsigned int j = 0; j < width; j++)
		{
			if (boardData[i][j] != mustHaveData[i][j]) same = false;
		}
	}
	return same;
}
```



Εικόνα 7.1 – Αποτελέσματα ελέγχου ορθότητας των αρχείων εξόδου

Τέλος, για τον έλεγχο της ορθότητας όλων των αρχείων εκτελέστηκε ένα batch file, που εκτελεί το πρόγραμμα για 10 διαφορετικές αρχικές συνθήκες και συγκρίνει τις εξόδους με τα σωστά αποτελέσματα. Τα αποτελέσματα παρουσιάζονται στην Εικόνα 7.1

7.2 Μέτρηση χρόνου εκτέλεσης

Το πρόγραμμα αναπτύχθηκε σε Visual Studio 2015 σε υπολογιστή με λειτουργικό σύστημα Microsoft Windows 10, επεξεργαστή CPU Intel(R) Core(TM) i5–4460 στα 3,20GHz και GPU Intel HD Graphics 4600. Η κάρτα γραφικών υποστηρίζει OpenCL C έκδοση 1.2 και OpenGL 4.0. Περισσότερα χαρακτηριστικά της κάρτας γραφικών παρουσιάζονται στο παράρτημα. Για την εκτέλεση του προγράμματος χρειάζεται να δοθεί το εκτελέσιμο πρόγραμμα gol.exe και δίπλα οι παρακάτω παράμετροι :

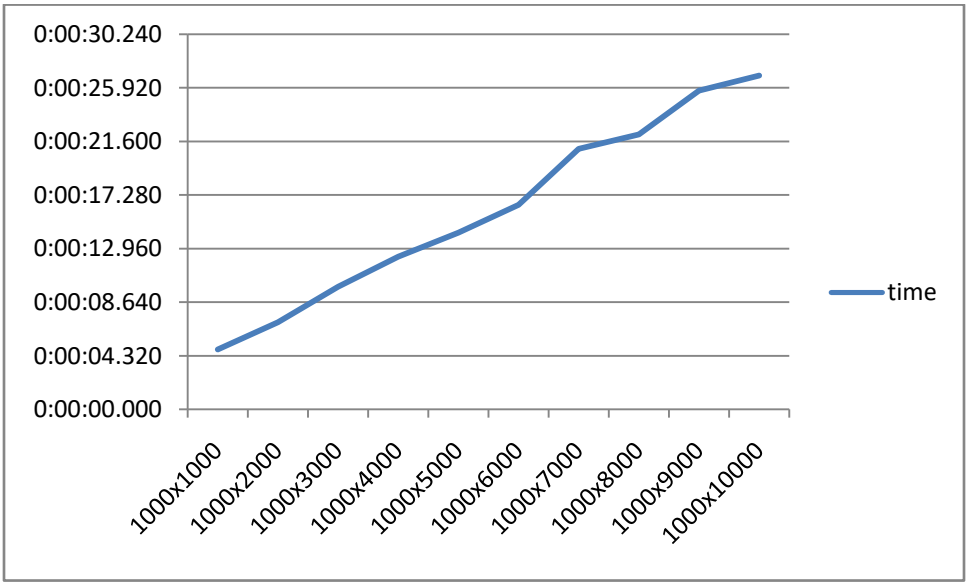
- --mode ocl : καθορίζουμε ότι το πρόγραμμα θα εκτελεστεί με OpenCL
- --device gpu: καθορίζουμε ότι η επιλεγμένη συσκευή είναι η GPU
- --measure: καθορίζουμε ότι το πρόγραμμα θα εμφανίσει χρόνους εκτέλεσης
- --load "step1000_in_250generations\random10000_in.gol": καθορίζουμε το αρχείο εισόδου με τις αρχικές θέσεις των κυττάρων
- --save "output/random1000_step250_ocl_gpu.gol": καθορίζουμε το αρχείο όπου θα αποθηκευτούν τα αποτελέσματα
- --generations 250 : καθορίζουμε τον μέγιστο αριθμό γενεών που θα εκτελεστεί το πρόγραμμα
- --must_out "step1000_out_250generations\random10000_out.gol": καθορίζουμε το αρχείο με τα σωστά αποτελέσματα μετά από 250 γενεές

Οι χρόνοι εκτέλεσης του προγράμματος για τις διαφορετικές αρχικές συνθήκες παρουσιάζονται παρακάτω (Πίνακας 7.1)

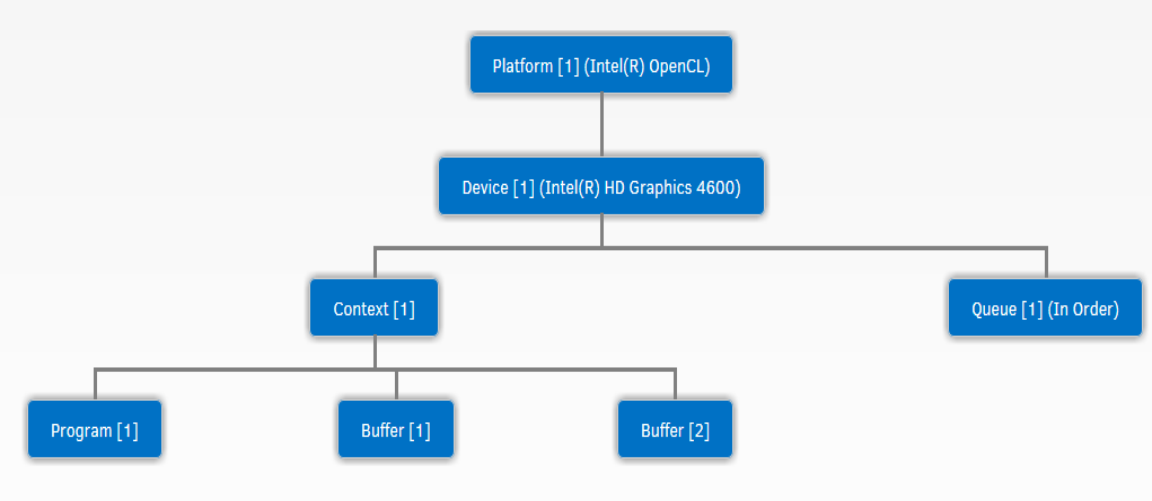
Πίνακας 7.1 – Χρόνοι εκτέλεσης (sec) για το παιχνίδι της ζωής με τη συνεργασία OpenCL/OpenGL για 250 generations

Συνολικά κύτταρα	1ο	2ο	3ο	4ο	5ο	6ο	7ο	8ο	9ο	10ο	**average**
1k	04.793	04.850	04.858	04.841	05.177	04.946	04.861	04.398	04.186	04.473	**04.738**
2k	07.704	07.710	07.807	07.794	07.684	07.615	06.979	06.693	06.719	07.099	**07.380**
3k	10.395	10.501	10.900	10.540	10.411	09.870	09.650	09.120	09.106	09.577	**10.007**
4k	13.273	13.134	13.199	14.064	13.100	12.132	11.923	11.492	11.455	11.899	**12.567**
5k	18.116	15.762	15.639	17.085	15.666	14.493	14.371	13.711	13.787	14.300	**15.293**
6k	18.290	18.293	18.324	20.464	18.273	16.862	16.421	15.893	15.873	16.837	**17.553**
7k	21.957	21.997	22.636	24.526	21.839	20.078	19.923	19.111	19.568	20.600	**21.223**
8k	22.918	23.456	23.689	26.332	23.356	21.465	21.306	20.421	20.895	22.977	**22.681**
9k	25.899	25.625	26.242	27.479	26.210	23.545	23.506	22.598	23.497	24.449	**24.905**
10k	28.223	28.660	28.864	33.622	28.543	26.018	26.008	24.952	26.074	27.336	**27.830**

Παρατηρούμε ότι όσο αυξάνεται το πλήθος των κυττάρων αυξάνεται και ο χρόνος εκτέλεσης του προγράμματος. Για περισσότερη ανάλυση του χρόνου εκτέλεσης των τμημάτων κώδικα του προγράμματος, χρησιμοποιήσαμε το Intel SDK – Code Builder for OpenCL (extension για το Visual Studio).



Εικόνα 7.2 – Διάγραμμα χρόνου εκτέλεσης για διαφορετικές τιμές εισόδου



Εικόνα 7.3 – Διαγραμματική αναπαράσταση των OpenCL αντικειμένων με το Intel SDK

7.3 Σύγκριση με το αρχικό πρόγραμμα

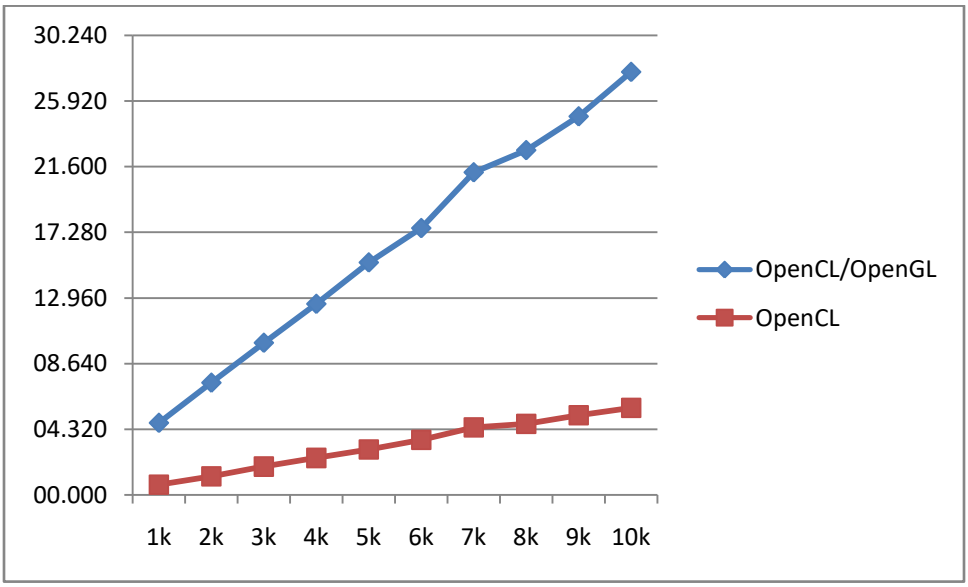
Το αρχικό πρόγραμμα (βλ. Κεφάλαιο 4.3) υλοποιεί το παιχνίδι της ζωής του Conway με OpenCL, χωρίς την παρουσία γραφικών. Οι χρόνοι εκτέλεσης με τις ίδιες συνθήκες παρουσιάζονται παρακάτω (Πίνακας 7.2). Παρατηρούμε ότι για τον υπολογισμό της επόμενης κατάστασης οι χρόνοι είναι σαφώς πιο μικροί ακόμα και όταν το μέγεθος του προβλήματος είναι μεγάλο.

Πίνακας 7.2 - Χρόνοι εκτέλεσης (sec) για το παιχνίδι της ζωής με OpenCL για 250 generation με υπολογισμό σε GPU

Συνολικά κύτταρα	1ο	2ο	3ο	4ο	5ο	6ο	7ο	8ο	9ο	10ο	**average**
1k	00.659	00.683	00.689	00.682	00.677	00.692	00.691	00.662	00.757	00.714	**00.691**
2k	01.214	01.201	01.219	01.224	01.229	01.222	01.337	01.212	01.268	01.266	**01.239**
3k	01.840	01.859	01.871	01.849	01.861	01.866	01.892	01.894	01.925	01.921	**01.878**
4k	02.427	02.418	02.415	02.484	02.426	02.426	02.459	02.413	02.528	02.458	**02.445**
5k	02.996	02.978	02.971	02.984	02.972	02.999	03.052	03.035	03.053	02.969	**03.001**
6k	03.592	03.776	03.622	03.709	03.572	03.614	03.575	03.564	03.688	03.588	**03.630**
7k	04.425	04.429	04.423	04.438	04.428	04.462	04.450	04.457	04.515	04.516	**04.454**
8k	04.619	04.739	04.595	04.707	04.672	04.670	04.647	04.660	04.782	04.773	**04.686**
9k	05.286	05.160	05.145	05.152	05.406	05.217	05.263	05.281	05.380	05.235	**05.252**
10k	05.833	05.828	05.672	05.675	05.729	05.661	05.707	05.705	05.762	05.720	**05.729**

Πίνακας 7.3 - Χρόνοι εκτέλεσης (sec) συνοπτικά για το παιχνίδι της ζωής με OpenCL/OpenGL σε σύγκριση με το αρχικό πρόγραμμα με OpenCL.

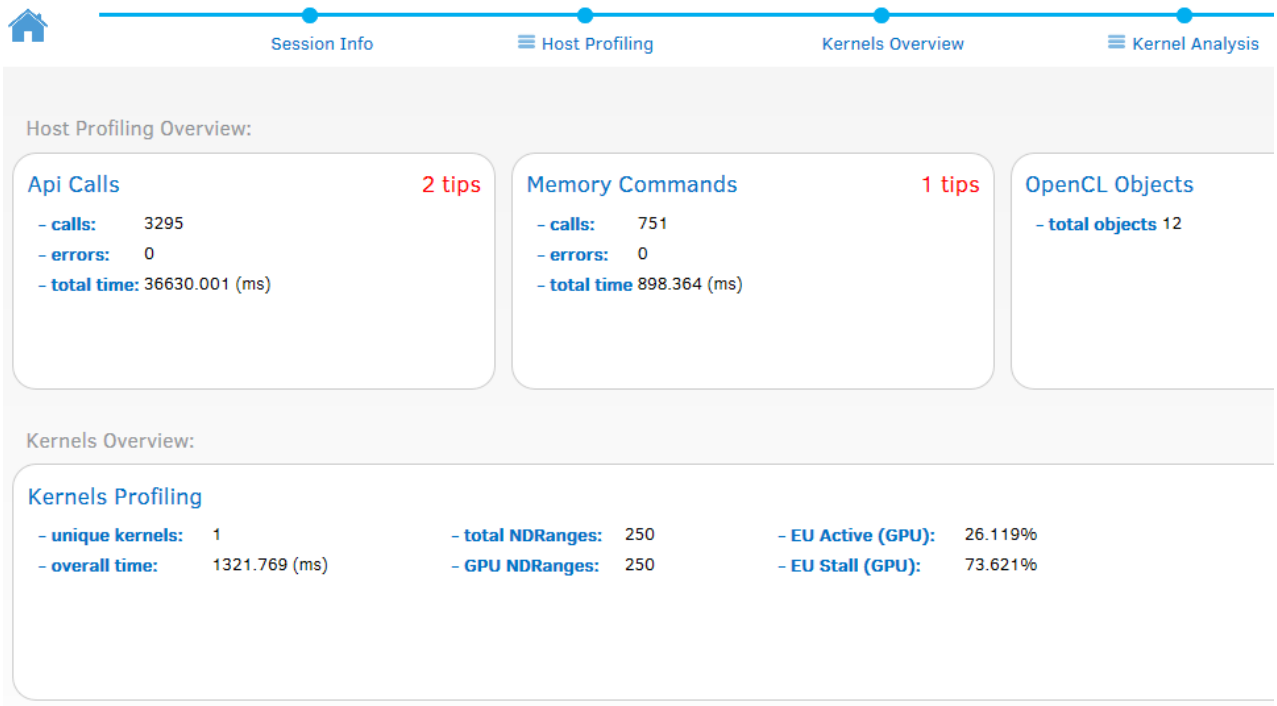
Συνολικά κύτταρα	OpenCL/OpenGL	OpenCL
1k	04.738	00.691
2k	07.380	01.239
3k	10.007	01.878
4k	12.567	02.445
5k	15.293	03.001
6k	17.553	03.630
7k	21.223	04.454
8k	22.681	04.686
9k	24.905	05.252
10k	27.830	05.729



Εικόνα 7.4 – Διάγραμμα χρόνων εκτέλεσης για διαφορετικές τιμές εισόδου σε σύγκριση με το αρχικό πρόγραμμα με OpenCL

7.4 Ανάλυση του προγράμματος σε OpenCL επίπεδο

Στην πρώτη ανάλυση της εφαρμογής, τα αποτελέσματα που πήραμε αφορούν την OpenCL. Η GPU με την OpenCL εργάστηκε περίπου 26% σε υπολογισμούς (Εικόνα 7.5) ενώ αναφέρεται ότι περίπου 73% έμεινε σταματημένη.



Εικόνα 7.5 – Γενικές πληροφορίες από το SDK της Intel για την OpenCL.

Παρακάτω, στην Εικόνα 7.6, φαίνεται ο συνολικός χρόνος κάθε OpenCL εντολής σε φθίνουσα σειρά. Η εντολή που «σπαταλάει» περισσότερο χρόνο είναι η εντολή συγχρονισμού clFinish(). Οι εντολές που αφορούν την διαχείριση buffer φαίνονται στην Εικόνα 7.7.

Api Calls: [Data Table] Graphical View

Api Name	Count	# Errors	Total Duration (ms)	Avg Duration (ms)
+ clFinish	250	0	36271.586	145.086
+ clBuildProgram	1	0	142.682	142.682
+ clCreateCommandQueue	1	0	54.022	54.022
+ clCreateContext	1	0	12.445	12.445
+ clEnqueueReadBuffer	1	0	9.582	9.582
+ clReleaseCommandQueue	1	0	4.553	4.553
+ clCreateBuffer	1	0	2.588	2.588
+ clEnqueueReleaseGLObjects	250	0	69.926	0.28
+ clReleaseContext	1	0	0.247	0.247
+ clCreateKernel	1	0	0.196	0.196
+ clReleaseKernel	1	0	0.145	0.145
+ clEnqueueNDRangeKernel	250	0	35.756	0.143
+ clCreateFromGLBuffer	1	0	0.073	0.073
+ clEnqueueCopyBuffer	250	0	9.904	0.04
+ clEnqueueAcquireGLObjects	250	0	6.659	0.027
+ clCreateProgramWithSource	1	0	0.018	0.018
+ clGetPlatformIDs	2	0	0.033	0.016
+ clReleaseProgram	1	0	0.008	0.008
+ clReleaseMemObject	1002	0	7.21	0.007
+ clSetKernelArg	4	0	0.03	0.007
+ clGetDeviceInfo	6	0	0.018	0.003
+ clRetainMemObject	1000	0	2.305	0.002
+ clReleaseDevice	4	0	0.005	0.001

Εικόνα 7.6 – Χρόνοι εκτέλεσης των εντολών της OpenCL από το SDK της Intel

Command Name	Count	# Errors	Total Duration (ms)	Avg Duration (ms)
+ CL_COMMAND_READ_BUFFER	1	0	6.025	6.025
+ CL_COMMAND_COPY_BUFFER	250	0	890.965	3.564
+ CL_COMMAND_ACQUIRE_GL_OBJECTS	250	0	0.826	0.003
+ CL_COMMAND_RELEASE_GL_OBJECTS	250	0	0.548	0.002

Εικόνα 7.7 – Χρόνοι εκτέλεσης των εντολών της OpenCL που αφορούν μνήμη από το SDK της Intel

Όπως παρατηρούμε, η εντολή που σπαταλάει περισσότερο χρόνο είναι η CL_COMMAND_COPY_BUFFER. Την χρησιμοποιούμε κάθε φορά που καλείται ο kernel για να αντιγράψουμε τα αποτελέσματα από τον SwapBuffer στον αρχικό buffer data και να υπολογιστεί εκ νέου η κατάσταση των κυττάρων. Η εντολή CL_COMMAND_READ_BUFFER σπαταλάει επίσης πολύ χρόνο και χρησιμοποιείται μία φορά για να αντιγράψουμε τα δεδομένα του buffer της GPU πίσω στην CPU. Παρακάτω, φαίνεται η ανάλυση που αφορά τον kernel. Όπως προαναφέρθηκε, οι υπολογισμοί έγιναν σε ποσοστό 26% .

Kernels Overview: [Data Table] Graphical View

Kernel Name	Global Work Size	Local Work Size	Device Type	Count	Total Duration (ms)	Avg Duration (ms)
+ Kernel [1] (GoI_All)	(3000000)		GPU	250	1321.769	5.287

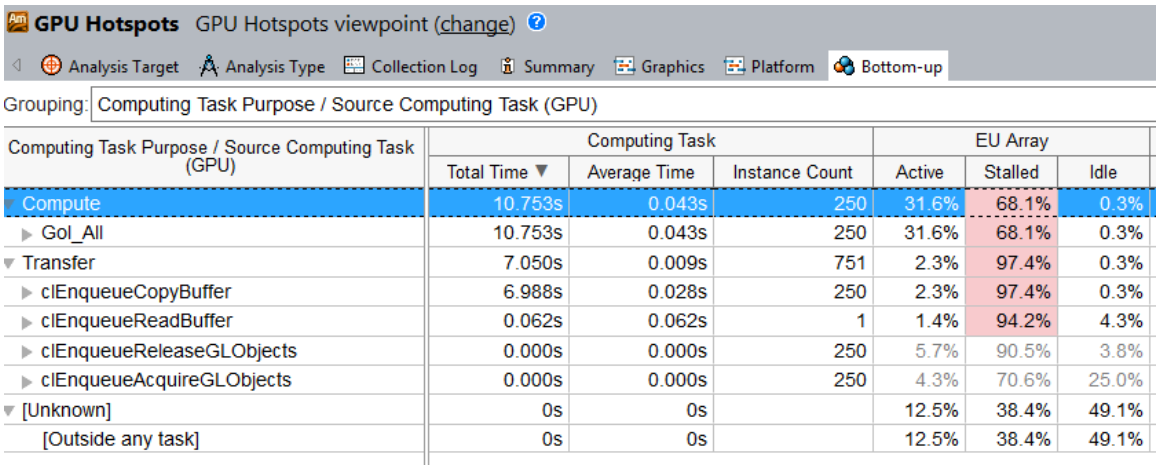
	EU Active (%)	EU Stall (%)	EU Idle (%)
[...]	26.12%	73.62%	0.26%

Εικόνα 7.8 - Χρόνοι εκτέλεσης και ποσοστό εργασίας του kernel από το SDK της Intel

7.5 Ανάλυση του προγράμματος με βάση την GPU

Σε ανάλυση που αφορά γενικότερα την κάρτα γραφικών (Vtune Amplifier της Intel), η GPU για υπολογισμούς εργάστηκε περίπου 31%, για μεταφορά δεδομένων περίπου 2% ενώ για την απεικόνιση των δεδομένων στην οθόνη περίπου το 12%. Ένα μεγάλο ποσοστό σύμφωνα με την ανάλυση η GPU παραμένει σταματημένη ή άεργη. (Εικόνα 7.9). Αυτό οφείλεται στην χρονοβόρα εντολή clFinish() που χρησιμοποιείται για

συγχρονισμό καθώς και στις εντολές διαχείρισης δεδομένων του buffer όπως clEnqueueCopyBuffer και clEnqueueReadBuffer.

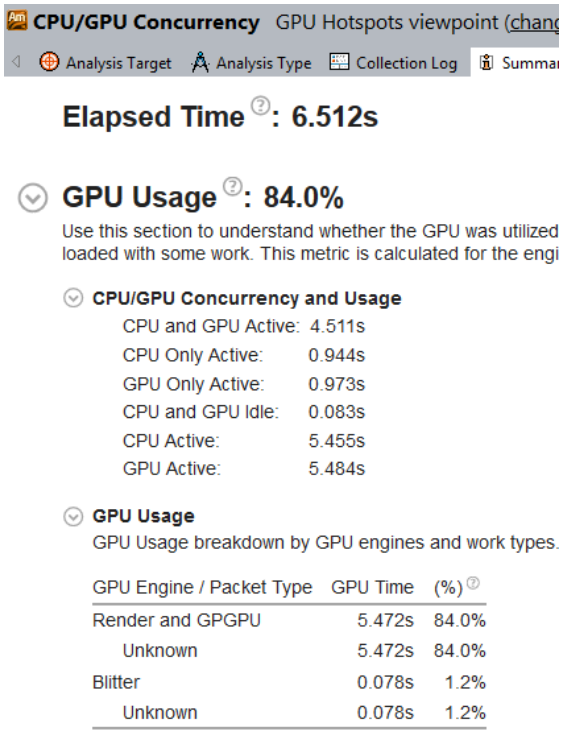


Computing Task Purpose / Source Computing Task (GPU)	Computing Task			EU Array		
	Total Time ▼	Average Time	Instance Count	Active	Stalled	Idle
Compute	10.753s	0.043s	250	31.6%	68.1%	0.3%
GoI_All	10.753s	0.043s	250	31.6%	68.1%	0.3%
Transfer	7.050s	0.009s	751	2.3%	97.4%	0.3%
clEnqueueCopyBuffer	6.988s	0.028s	250	2.3%	97.4%	0.3%
clEnqueueReadBuffer	0.062s	0.062s	1	1.4%	94.2%	4.3%
clEnqueueReleaseGLObjects	0.000s	0.000s	250	5.7%	90.5%	3.8%
clEnqueueAcquireGLObjects	0.000s	0.000s	250	4.3%	70.6%	25.0%
[Unknown]	0s	0s		12.5%	38.4%	49.1%
[Outside any task]	0s	0s		12.5%	38.4%	49.1%

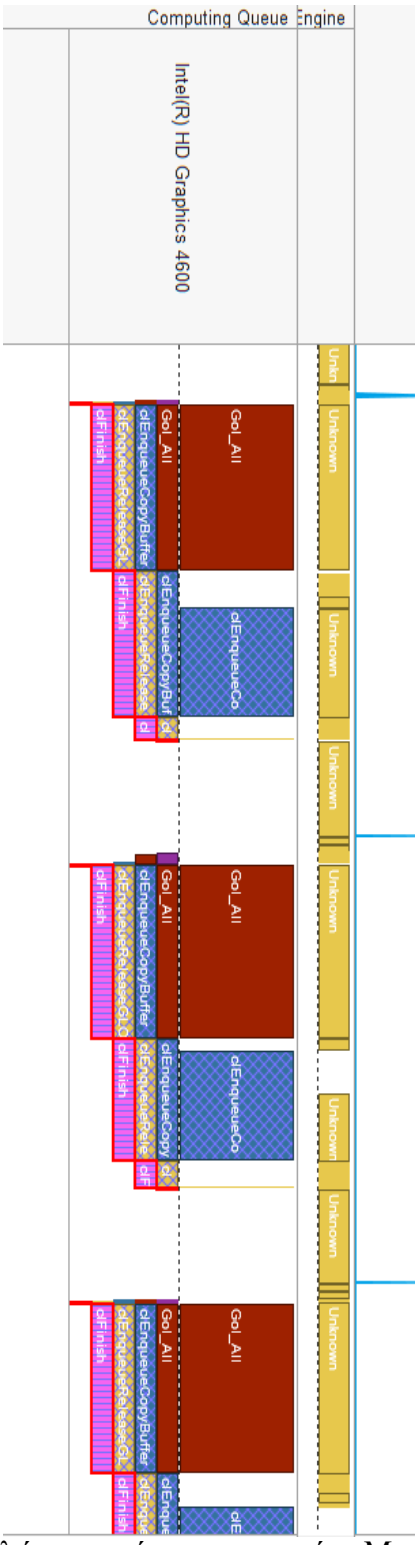
Εικόνα 7.9 – Πληροφορίες σχετικά με χρόνους των εντολών σε κατηγορίες από το Intel VTune Amplifier 2018

7.6 Ανάλυση της GPU/CPU

Σε ανάλυση που αφορά την σχέση CPU/GPU, η κάρτα γραφικών χρησιμοποιήθηκε σε ποσοστό 84% όπως φαίνεται στην Εικόνα 7.10. Ένα στιγμιότυπο από την ουρά εντολών της κάρτας γραφικών φαίνεται στην Εικόνα 7.11.



Εικόνα 7.10 – Χρήση GPU σε σύγκριση με την CPU.



Εικόνα 7.11 - Ουρά εντολών της κάρτας γραφικών. Με την σειρά εκτελείται ο kernel (κόκκινο), η εντολή clEnqueueCopyBuffer (Μπλε), η εντολή clEnqueueReleaseGL(κίτρινο) και στη συνέχεια η clFinish (ροζ).

ΚΕΦΑΛΑΙΟ 8

8 Συμπεράσματα και μελλοντικές κατευθύνσεις

Στην παρούσα εργασία, παρουσιάσαμε γραφικά το παιχνίδι της ζωής με συνεργασία των δύο τεχνολογιών πάνω σε GPU, την OpenCL και την OpenGL. Θα μπορούσαν μελλοντικά να γίνουν βελτιώσεις τόσο σε υπολογιστικό επίπεδο όσο και σε επίπεδο εμφάνισης.

Σε υπολογιστικό επίπεδο μπορεί να μειωθεί η απαιτούμενη μνήμη που θα μεταφέρεται στην GPU. Κάθε μεμονωμένο κύτταρο έχει δύο καταστάσεις που τις αναπαριστούμε με τους ακέραιους αριθμούς 0 και 1 που σημαίνει για πληροφορία του 1bit με την μορφή ακεραίου δεσμεύουμε 8-bit. Μια βελτίωση του κώδικα υλοποίησης θα ήταν η αποθήκευση της κατάστασης των κυττάρων σε 1bit ανά κύτταρο και όχι σε 8bit. Θα μπορούσε να μειώσει την απαιτούμενη μνήμη περίπου 8 φορές. Επίσης, βελτιώσεις θα μπορούσαν να γίνουν στον κώδικα σύμφωνα με τα διαγνωστικά εργαλεία που παρέχει η Intel. Μια από αυτές αφορά την αντικατάσταση της εντολής clEnqueueReadBuffer με την εντολή clEnqueueMapBuffer, καθώς η αντιστοίχιση διεύθυνσης είναι σαφώς πιο γρήγορη από την αντιγραφή των δεδομένων. Η δεύτερη βελτίωση αφορά την απόδοση της OpenCL. Ο kernel εκτελείται χωρίς να ορίσουμε local work size (NULL). Στο μέλλον θα μπορούσαμε να εκτελέσουμε τον κώδικα ορίζοντας διαφορετικές τοπικές αποικίες και να επιλέξουμε το local group που αποδίδει καλύτερα με την συγκεκριμένη κάρτα γραφικών. Η ανάλυση του kernel στην Εικόνα 8.1 δείχνει ότι η καλύτερη ρύθμιση για τα local group size είναι L(64,0,0).

Execution View: Execution [Advanced] Graphical View

Best Configuration: G(3000000,0,0) - L(64,0,0) - median (ms): 4.27488

Gx	Gy	Gz	Lx	Ly	Lz	Iterations	Total (ms)	Queue (ms)	Submit (ms)	Execution (ms)
+ 3000000	0	0	64	0	0	1	4.27488	0.0112	0.06504	4.0836
+ 3000000	0	0	0	0	0	1	5.1824	0.08224	0.63576	4.314
+ 3000000	0	0	32	0	0	1	5.0464	0.01264	0.0712	4.55304
+ 3000000	0	0	16	0	0	1	5.15392	0.0268	0.08624	4.87048
+ 3000000	0	0	8	0	0	1	6.42848	0.0108	0.12208	6.15304
+ 3000000	0	0	4	0	0	1	9.37184	0.00928	0.10736	9.14512
+ 3000000	0	0	2	0	0	1	14.416	0.0264	0.12896	14.1694
+ 3000000	0	0	1	0	0	1	28.6493	0.01	0.10248	28.4181

Εικόνα 8.1 – Ανάλυση kernel για εύρεση καλύτερου local group

Σε επίπεδο εμφάνισης θα μπορούσαμε να προσθέσουμε λειτουργίες αλληλεπίδρασης με τον χρήστη, όπως μενού επιλογών καθώς και λειτουργία zoom ώστε να εστιάζεται η προβολή σε συγκεκριμένη περιοχή κυττάρων. Έτσι, θα φαίνεται με λεπτομέρεια και η κίνηση των κυττάρων τοπικά και συνολικά με όλο το αρχικό πλήθος των κυττάρων.

Βιβλιογραφία

1. **Wikipedia, the free encyclopedia.** Conway's Game of Life. *Wikipedia.* [Online] May 2018. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

2. **Education, Stanford.** web.stanford.edu. *Conway's Game of Life.* [Ηλεκτρονικό] cs.stanford.edu. http://web.stanford.edu/~cdebs/GameOfLife/.

3. **Melissa Gymrek.** web.mit.edu. *Conway's Game of Life.* [Ηλεκτρονικό] May 2010. http://web.mit.edu/sp.268/www/2010/lifeSlides.pdf.

4. **Paul Callahan .** What is the Game of Life? *math.com - The world of Math online.* [Ηλεκτρονικό] math.com, 2005. [Παραπομπή: 8 September 2018.] http://www.math.com/students/wonders/life/life.html.

5. **Stanford University.** Conway's Game of Life. *Stanford University.* [Ηλεκτρονικό] [Παραπομπή: 8 September 2018.] http://web.stanford.edu/~cdebs/GameOfLife/.

6. **Guan, Puhua.** Cellular Automaton Public-Key Cryptosystem. *Complex Systems.* 1987, Τόμ. 1, σσ. 51-57.

7. **WOLFRAM, STEPHEN.** Random Sequence Generation by Cellular Automata. *ADVANCES IN APPLIED MATHEMATICS.* 7, 1986, Τόμ. 7, 2, σσ. 123-252.

8. **Howard, Toby.** OpenGL Manual. *http://syllabus.cs.manchester.ac.uk/.* [Ηλεκτρονικό] http://syllabus.cs.manchester.ac.uk/ugt/2017/COMP27112/doc/OpenGL-manual.pdf.

9. **Wikipedia.** OpenGL. *Wikipedia The Free Encyclopedia.* [Ηλεκτρονικό] May 2018. https://en.wikipedia.org/wiki/OpenGL.

10. **Khronos Group.** OpenGL Getting Started. *Khronos Group.* [Ηλεκτρονικό] February 2018. https://www.khronos.org/opengl/wiki/Getting_Started.

11. **Theodoros Alexandropoulos-MediaLab.** Εισαγωγή στην OpenGL. *MediaLab.* [Ηλεκτρονικό]
http://www.medialab.ntua.gr/education/ComputerGraphics/OpenGL_Lectures/01-Introduction.pdf.

12. **Shreiner, Dave, et al.** The Official Guide to Learning OpenGL, Version 4.3. *OpenGL Programming Guide.* 2013.

13. **OpenGL.** 2.3 glutInitDisplayMode. *OpenGL The Industry's Foundation for High Performance Graphics.* [Ηλεκτρονικό] 1996. https://www.opengl.org/resources/libraries/glut/spec3/node12.html.

14. **Freeglut.** The Open-Source OpenGL Utility Toolkit. *Freeglut The Free OpenGL Utility Toolkit.* [Ηλεκτρονικό] Janouary 2013. http://freeglut.sourceforge.net/docs/api.php.

15. **Easy Lessons.** Game Life (C++ OpenGL (GLUT)). *www.youtube.com.* [Ηλεκτρονικό] 11 November 2017. [Παραπομπή: 20 September 2018.] https://www.youtube.com/watch?v=NPvwGh2ucPk.

16. **Gaster, Benedict R.** AMD Products group. *slideplayer*. [Ηλεκτρονικό] http://slideplayer.com/slide/3390387/.

17. **Pertiller, David.** CONWAY'S GAME OF LIFE PARALLELIZED. *David Pertiller.* [Ηλεκτρονικό] https://www.pertiller.tech/public-projects/open-source-projects.

18. **Howes, Benedict R. Gaster and Lee.** The OpenCL C++ Wrapper API. *https://www.khronos.org/.* [Ηλεκτρονικό] https://www.khronos.org/registry/OpenCL/specs/opencl-cplusplus-1.2.pdf.

19. **Fišer, Marek.** Conway's Game of Life on GPU using CUDA. *MarekFiser.com.* [Ηλεκτρονικό] March 2013. http://www.marekfiser.com/Projects/Conways-Game-of-Life-on-GPU-using-CUDA.

20. **Shevtsov, Maxim.** OpenCL™ and OpenGL* Interoperability Tutorial. *Intel Sofware Developer Zone.* [Online] Intel Corporation, April 28, 2014. https://software.intel.com/en-us/articles/opencl-and-opengl-interoperability-tutorial.

21. **J. Kowalik, T. Puźniakowski.** Using OpenCL: Programming Massively Parallel Computers. *Using OpenCL: Programming Massively Parallel Computers.* Amsterdam : IOS Press, 2012, 3, σ. 203.

22. **Bailey, Mike.** Oregon State University OSU - Parallel Programming CS 475. *Oregon State University.* [Ηλεκτρονικό] May 2017. http://web.engr.oregonstate.edu/~mjb/cs475/Handouts/opencl.opengl.vbo.1pp.pdf.

23. **Pertiller, David.** Mobiletainment/Conways-Game-of-Life-Parallelized. *github.com.* [Ηλεκτρονικό] 2012. https://github.com/Mobiletainment/Conways-Game-of-Life-Parallelized.

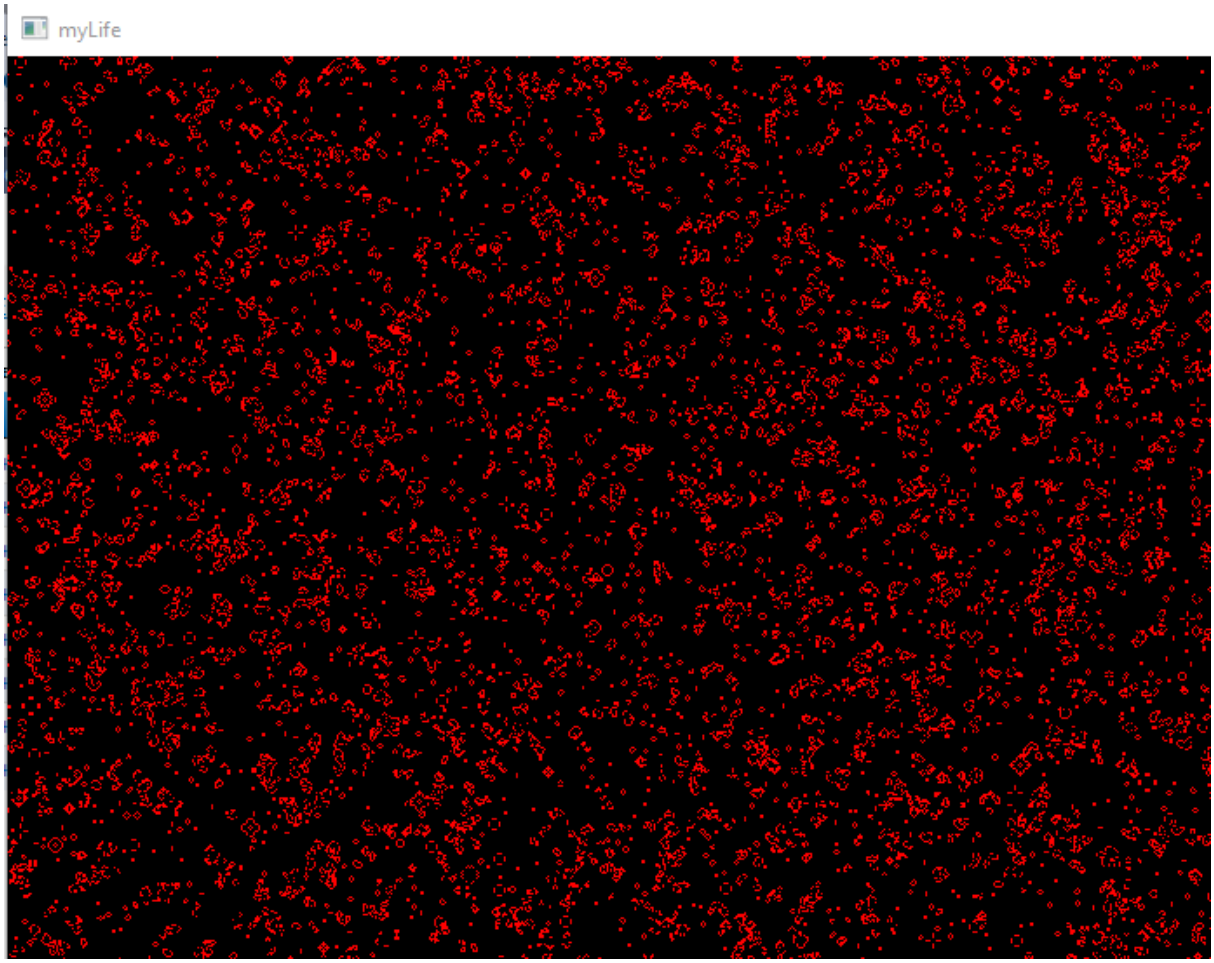
24. **Αγγελος, Γκάνιας.** Πτυχιακή εργασία με Τίτλο "Γραφικά Υπολογιστών". *Ιδρυματικό Αποθετήριο TEI Ηπείρου.* [Ηλεκτρονικό] 2016. http://apothetirio.teiep.gr/xmlui/bitstream/handle/123456789/5432/1425.pdf?sequence=1.

25. **Scarpino, Matthew.** *OpenCL in Action.* s.l. : Manning Publications, 2012. ISBN 9781617290176.

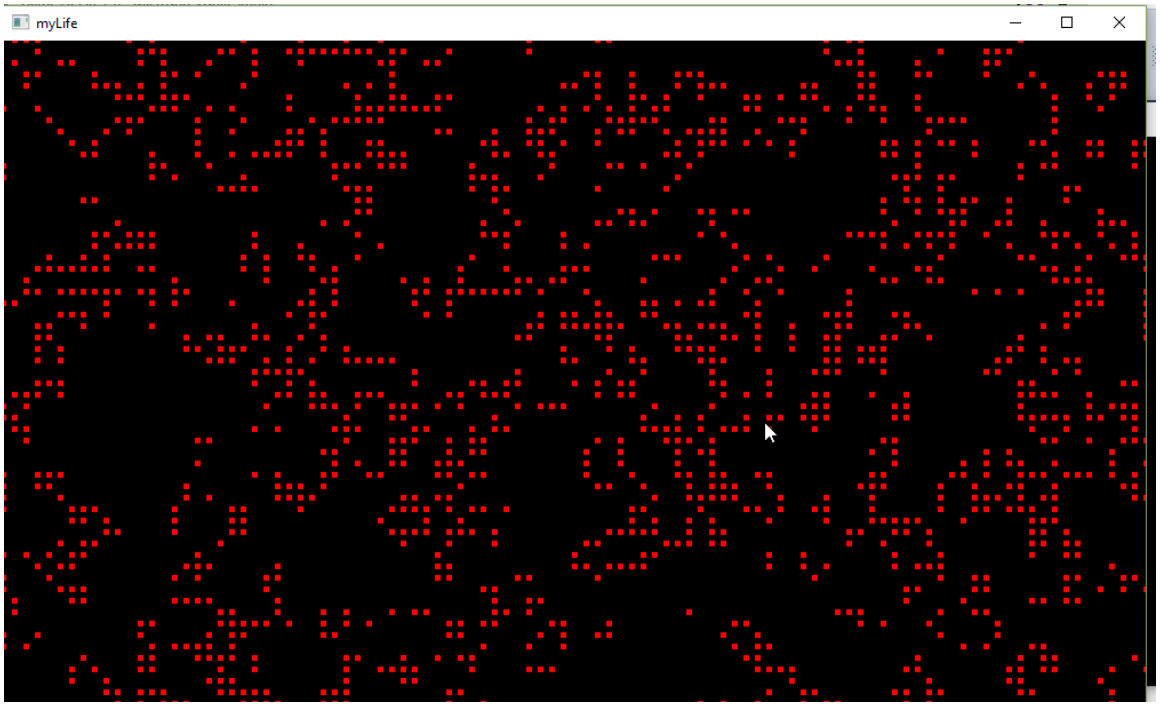
Α. Παράρτημα

Παρακάτω, παρουσιάζονται εικόνες από την υλοποίηση της εργασίας καθώς και ο κώδικας υλοποίησης με OpenGL.

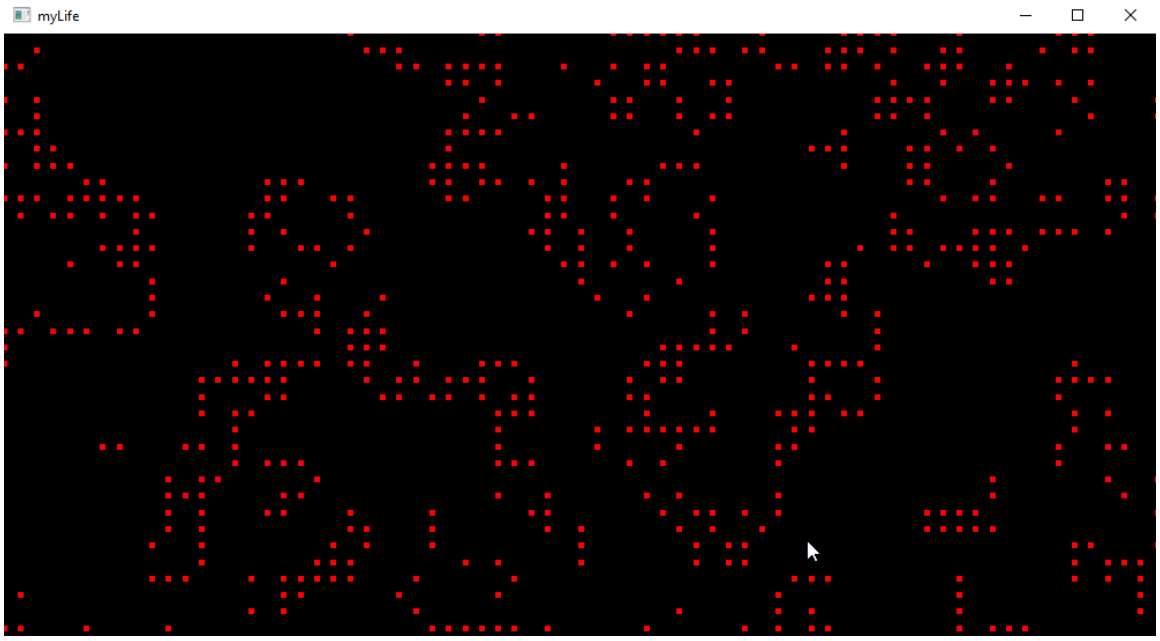
Εικόνες



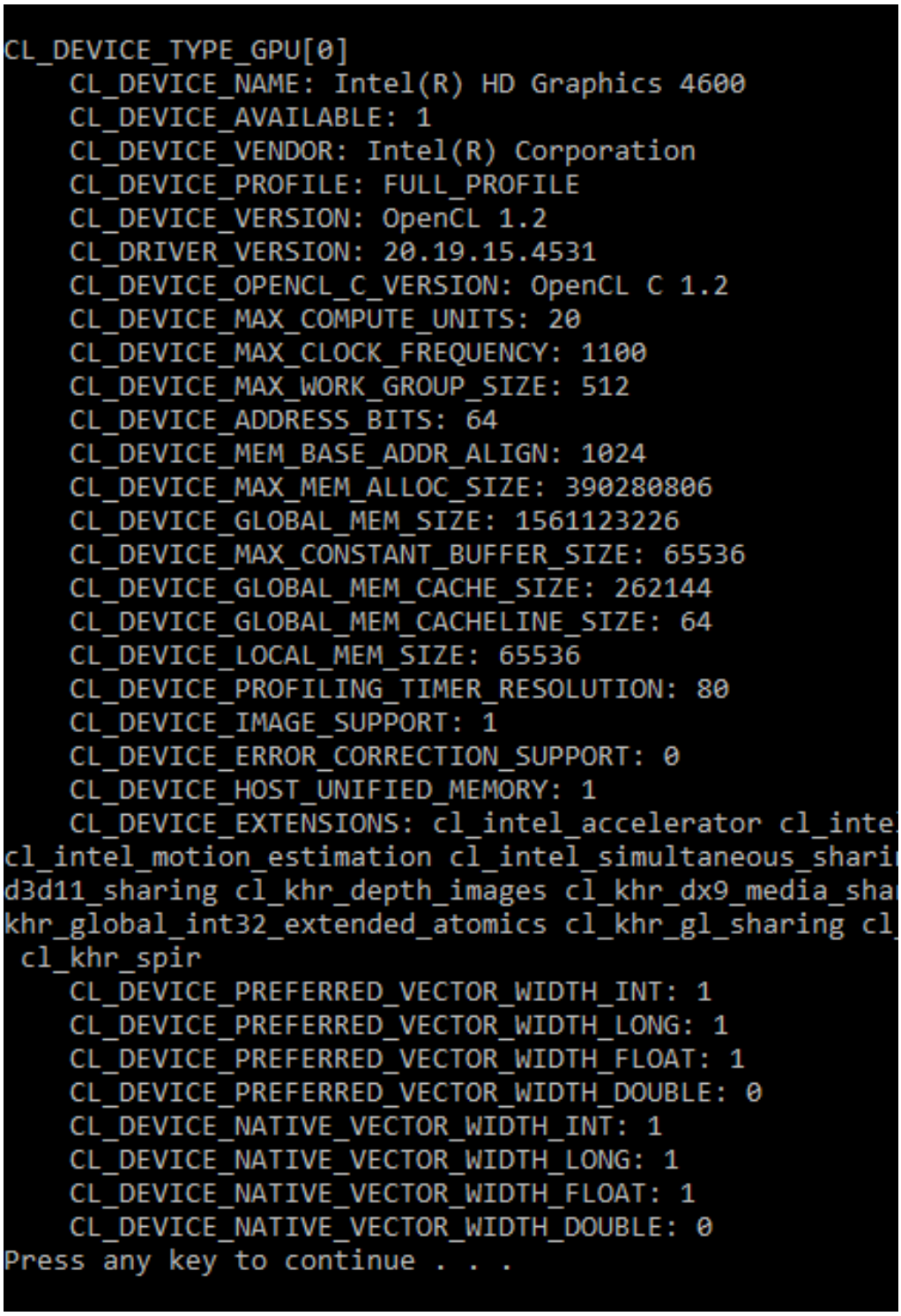
Εικόνα Α.0.1 - Γραφική αναπαράσταση των κυττάρων στο Παιχνίδι της ζωής με OpenCL/ OpenGL.



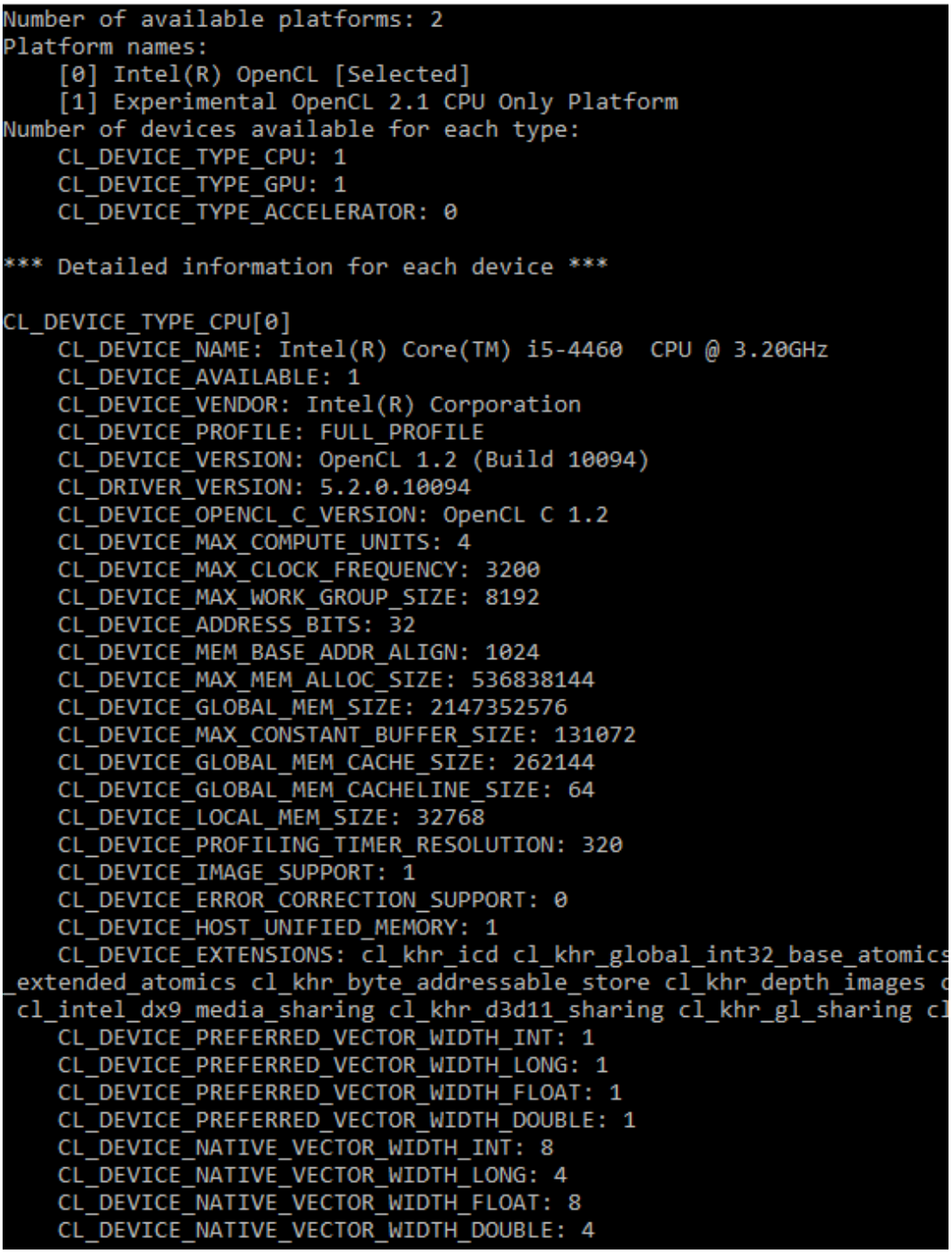
Εικόνα Α.0.2 - Γραφική αναπαράσταση των κυττάρων στο Παιχνίδι της ζωής με OpenCL/ OpenGL.



Εικόνα Α.0.3 - Γραφική αναπαράσταση των κυττάρων στο Παιχνίδι της ζωής με OpenCL/ OpenGL.



Εικόνα Α.0.4 - Χαρακτηριστικά της GPU



Εικόνα Α.0.5 - Χαρακτηριστικά της CPU

Κώδικας υλοποίησης με τεχνολογία OpenGL

```
#include <GL/glut.h>

int   m_x, m_y;               // mouse x y
bool  m_down = false;

int   play = true;          //auto play
int   size_ = 2;          //size of cell

const int   X = 300;
const int   Y = 200;


int w = X*size_;
int h = Y*size_;

struct P
{
      bool life;
      int next;
} p[X][Y];

void display()
{
      glClear(GL_COLOR_BUFFER_BIT);
      glColor3ub(255, 255, 255);
      glPointSize(size_);
      glBegin(GL_POINTS);
      for (int y = 0; y < Y; ++y)
            for (int x = 0; x < X; ++x)
            if (p[x][y].life) glVertex2f(size_/2 + x*size_, size_/2+y*size_);


      if (m_down && m_x > 0 && m_y > 0 && m_x < X * size_ && m_y < Y*size_)

            p[m_x / size_][m_y / size_].life = 1;

      else
      {
            int x = m_x / size_;  int y = m_y / size_;
            glVertex2f(size_ / 2 + size_ * x, size_ / 2 + size_ * y);
      }

      glEnd();

      glutSwapBuffers(); //glFlush();
```

```
}

int left(int xx)
{
      if (xx == 0) xx = X - 1; else xx--;
      return xx;
}

int right(int xx)
{
      if (xx == X - 1) xx = 0; else xx++;
      return xx;
}


int up(int yy)
{
      if (yy == 0) yy = Y - 1; else yy--;
      return yy;
}

int down(int yy)
{
      if (yy == Y - 1) yy = 0; else yy++;
      return yy;
}

void update()
{

      for (int y = 0; y < Y; ++y) for (int x = 0; x < X; ++x)
      {
            int counter = 0;

            if (p[left(x)][y].life)    counter++;         //left
            if (p[right(x)][y].life)    counter++;         //right
            if (p[x][up(y)].life) counter++;        //up
            if (p[x][down(y)].life) counter++;      //down

            if (p[left(x)][up(y)].life) counter++;        //left  +  up
            if (p[right(x)][up(y)].life) counter++;         //right + down

            if (p[left(x)][down(y)].life) counter++;      //left + up
            if (p[right(x)][down(y)].life) counter++;      //right

            if (p[x][y].life) if (counter != 2 && counter != 3) p[x][y].next
= false; else p[x][y].next = true;
            if (!p[x][y].life) if (counter == 3) p[x][y].next = true; else
p[x][y].next = false;
```

```
      }
      for (int y = 0; y < Y; ++y) for (int x = 0; x < X; ++x)
            p[x][y].life = p[x][y].next;


}


void timer(int = 0)
{

      if (GetAsyncKeyState('1'))  play = 1; // 1 2 3 это клавиши ))
      if (GetAsyncKeyState('2'))  play = 0;
      if (GetAsyncKeyState('3'))
for (int y = 0; y < Y; ++y)
for (int x = 0; x < X; ++x)
            {
                  p[x][y].next = 0; p[x][y].life = 0;
            } //clear
      if (play) update();
      display();
      glutTimerFunc(10, timer, 0);
}

void mouse(int button, int state, int ax, int ay)
{
      m_y = ay;
      m_x = ax;
      m_down = state == GLUT_DOWN;
}

void motion(int ax, int ay)
{
      m_x = ax;
      m_y = ay;
}

void mousewrite()
{

}

void motionpass(int ax, int ay)
{
      m_x = ax;
      m_y = ay;
}

int main(int argc, char **argv)
{
```

```
	for (int x = 0; x < X; x++)
		for (int y = 0; y < Y; y++)
		{
			p[x][y].life = rand() % 2;
			p[x][y].next = 0;
		}

	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
	glutInitWindowSize(w, h);
	glutInitWindowPosition(-10, -10);
	glutCreateWindow("Game of Life OpenGL");
	glClearColor(0, 0, 0, 1.0);
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	glOrtho(0, w, h, 0, -1, 1);
	glutDisplayFunc(display);
	timer();
	glutMouseFunc(mouse);
	glutMotionFunc(motion);
	glutPassiveMotionFunc(motionpass);
	mousewrite();
	glutMainLoop();
}
```