Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Πανεπιστήμιο Θεσσαλίας Βόλος 2013

[ΣΧΕΔΙΑΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΤΗΝ ΜΕΛΕΤΗ ΚΑΙ ΑΠΟΔΟΤΙΚΗ ΑΠΕΙΚΟΝΗΣΗ ΣΥΝΑΡΤΗΣΕΩΝ ΤΥΠΟΥ FRACTAL]

[SOFTWARE DESIGN AND IMPLEMENTATION FOR THE STUDY AND EFFICIENT IMAGING OF FRACTAL FUNCTIONS]

Διπλωματική Εργασία

του

Χρήστου Καλονάκη

Βόλος, Ιούλιος 2013

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ

ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΣΧΕΔΙΑΣΗ ΚΑΙ ΥΛΟΠΟΙΗΣΗ ΛΟΓΙΣΜΙΚΟΥ ΓΙΑ ΤΗΝ ΜΕΛΕΤΗ ΚΑΙ ΑΠΟΔΟΤΙΚΗ ΑΠΕΙΚΟΝΗΣΗ ΣΥΝΑΡΤΗΣΕΩΝ ΤΥΠΟΥ FRACTAL

SOFTWARE DESIGN AND IMPLEMENTATION FOR THE STUDY AND EFFICIENT IMAGING OF FRACTAL FUNCTIONS

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

Χρήστου Καλονάκη

Επιβλέποντες :

Παναγιώτα Τσομπανοπούλου	**Παναγιώτης Μποζάνης**
Επίκουρη Καθηγήτρια, Πανεπιστήμιο Θεσσαλίας	Αναπληρωτής Καθηγητής, Πανεπιστήμιο Θεσσαλίας

Εγκρίθηκε από την διμελή εξεταστική επιτροπή την 02/07/2013

(Υπογραφή)	*(Υπογραφή)*
.....	.....
ΚΥΡΙΟΣ ΕΠΙΒΛΕΠΩΝ	ΔΕΥΤΕΡΕΥΩΝ ΕΠΙΒΛΕΠΩΝ
Επίκουρη Καθηγήτρια Π.Θ.	Αναπληρωτής Καθηγητής Π.Θ.

Βόλος, Ιούλιος 2013

(Υπογραφή)

.....

Χρήστος Καλονάκης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών,

Πανεπιστημίου Θεσσαλίας

Στην οικογένειά μου και στους φίλους μου.

Ευχαριστίες

Με την περάτωση της παρούσας εργασίας, θα ήθελα να ευχαριστήσω θερμά τους επιβλέποντες της Διπλωματικής εργασίας κ. Παναγιώτα Τσομπανοπούλου και κ. Παναγιώτη Μποζάνη για την εμπιστοσύνη που επέδειξαν στο πρόσωπό μου, για την άριστη συνεργασία, τη συνεχή καθοδήγηση, τις ουσιώδεις υποδείξεις και παρεμβάσεις, που διευκόλυναν την εκπόνηση της Διπλωματικής εργασίας.

Επίσης θα ήθελα να εκφράσω την εκτίμησή μου για όλα τα παιδιά, με τα οποία συνεργάστηκα όλα αυτά τα χρόνια των προπτυχιακών σπουδών μου.

Τέλος, οφείλω ένα μεγάλο ευχαριστώ στην οικογένειά μου και στους φίλους μου για την αμέριστη υποστήριξη και την ανεκτίμητη βοήθεια που μου παρείχαν τόσο κατά την διάρκεια των σπουδών μου, όσο και κατά την εκπόνηση της Διπλωματικής εργασίας.

Χρήστος Καλονάκης
Βόλος, 2013

Περίληψη

Σκοπός της παρούσας Διπλωματικής εργασίας ήταν η ανάλυση ενός πλήρους λογισμικού απεικόνισης fractals για δισδιάστατες εικόνες, καθώς και των θεμάτων που προέκυψαν κατά τη διάρκεια ανάπτυξης της εφαρμογής.

Συγκεκριμένα, στις παρακάτω ενότητες αναλύονται διάφορες συναρτήσεις παραγωγής fractals, καθώς και τα Julia sets τους. Επίσης αναπτύσσεται η λειτουργία αλγορίθμων χρωματισμού της εικόνας, η δημιουργία της παλέτας των χρωμάτων, καθώς και άλλων χρωματικών λειτουργιών. Επιπρόσθετα επεξηγούνται διάφοροι μετασχηματισμοί της εικόνας καθώς και ένας αριθμός βελτιστοποιήσεων με έμφαση στην απόδοση της εφαρμογής. Τέλος καθορίζονται διάφοροι αλγόριθμοι επεξεργασίας της εικόνας, όπως για παράδειγμα Anti-Aliasing, Edge Detection κ.α.

Σε όλα τα στάδια της ανάλυσης της εφαρμογής, γίνεται χρήση ψευδοκώδικα και παρατίθεται ένας επαρκής αριθμός παραδειγμάτων εικόνων, με έμφαση στην επεξήγηση και στην κατανόηση της εφαρμογής.

Abstract

The purpose of this thesis, is the analysis of a complete fractal generating software, for two dimensional images and the issues that arose during the development of the application.

Specifically, in the following sections, we analyze a number of fractal producing functions along with their Julia sets. Moreover we analyze the coloring algorithms, the creation of the colored palette, and various coloring functions. Additionally, we analyze various image transformations along with a number of speed improvements that emphasize on performance of the application. Finally we define various image processing algorithms, such as Anti-Aliasing, Edge Detection, etc.

During the stages of the analysis of the application, we use pseudocode and a sufficient number of image examples, that emphasize on explanation and understanding of the application.

Περιεχόμενα

1. Εισαγωγή

Τον Απρίλιο του 2011, ανατέθηκε σε συμφοιτητές μου, η εκπόνηση εργασίας στο μάθημα Προγραμματισμός ΙΙ και συγκεκριμένα η δημιουργία μιας εφαρμογής που θα απεικόνιζε το Mandelbrot set. Μολονότι το συγκεκριμένο μάθημα το είχα ήδη περάσει σε προηγούμενο εξάμηνο, αποφάσισα να ασχοληθώ με το εν λόγω θέμα, επειδή αφ' ενός μεν είχα ιδιαίτερη κλίση στα μαθήματα προγραμματισμού και αφ' ετέρου το άγνωστο του θέματος μου κέντρισε το ενδιαφέρον, ώστε να προσπαθήσω να το προγραμματίσω και να εμπλουτίσω τις προγραμματιστικές μου γνώσεις και τεχνικές.

Έτσι ξεκίνησα ακολουθώντας τις απαιτήσεις της εργασίας και σιγά σιγά άρχισα να έχω το επιθυμητό αποτέλεσμα. Παρόλα αυτά οι απαιτήσεις αυτές ήταν μόνο η «κορυφή του παγόβουνου», καθόσον για να κατανοήσω κάποιες από τις έννοιες που πραγματεύονταν η εργασία, άρχισα να αναζητώ στο διαδίκτυο επιπλέον πληροφορίες. Μέρος αυτών των πληροφοριών με οδήγησαν στο να «κατεβάσω» διάφορα λογισμικά απεικόνισης του Mandelbrot set. Όμως με κατάπληξη παρατήρησα, πως το Mandelbrot set δεν ήταν το μοναδικό "fractal", αλλά υπήρχε άπειρος αριθμός συναρτήσεων, κυριολεκτικά, που μπορούσε να χρησιμοποιηθεί για τέτοιου είδους απεικονίσεις.

Όπως ήταν λογικό και αυτονόητο, η έντονη επιθυμία μου, να αναπτύξω τις προγραμματιστικές μου τεχνικές, με οδήγησε στην πρόσθεση περισσότερων συναρτήσεων απεικόνισης στην εφαρμογή. Αν και η προσπάθεια μου στέφτηκε με επιτυχία, ωστόσο αναρωτιόμουν για το τι άλλο θα μπορούσα ακόμα να προσθέσω. Τα λογισμικά απεικόνισης ήταν πλήρως παραμετροποιημένα, οπότε το ίδιο προσπάθησα να κάνω και εγώ με τη δική μου εφαρμογή.

Έτσι, επιχείρησα να αναπτύξω περαιτέρω την εφαρμογή, δηλαδή παρατηρώντας αλλά λογισμικά απεικόνισης προσπάθησα να αντιγράψω αρκετές από τις λειτουργίες τους, και χρησιμοποιώντας πληροφορίες και από το διαδίκτυο, κατέληξα στο να έχω γράψει πάνω από 23.500 γραμμές κώδικα σε Java (Απρίλιος 2011 – Φεβρουάριος 2013), με αποτέλεσμα όλη αυτή η προσπάθεια να αποτελέσει την παρούσα Διπλωματική εργασία.

Οι παρακάτω σελίδες θα προσπαθήσουν να ρίξουν λίγο φως στα διαφορετικά στάδια ανάπτυξης της εφαρμογής.

2. Ιστορικά στοιχεία – Πληροφορίες σχετικά με τα Fractals

Με τον διεθνή όρο fractal (φράκταλ, ελλ. μορφόκλασμα ή μορφοκλασματικό σύνολο) στα μαθηματικά, τη φυσική αλλά και σε πολλές επιστήμες ονομάζεται ένα γεωμετρικό σχήμα που επαναλαμβάνεται αυτούσιο σε άπειρο βαθμό μεγέθυνσης, κι έτσι συχνά αναφέρεται σαν "απείρως περίπλοκο". Το fractal παρουσιάζεται ως "μαγική εικόνα" που όσες φορές και να μεγεθυνθεί οποιοδήποτε τμήμα του θα συνεχίζει να παρουσιάζει ένα εξίσου περίπλοκο σχέδιο με μερική ή ολική επανάληψη του αρχικού. Χαρακτηριστικό επομένως των fractals είναι η λεγόμενη αυτο-ομοιότητα (self-similarity) σε κάποιες δομές τους, η οποία εμφανίζεται σε διαφορετικά επίπεδα μεγέθυνσης.

Τα fractals σε πολλές περιπτώσεις μπορεί να προκύψουν από τύπο που δηλώνει αριθμητική, μαθηματική ή λογική επαναληπτική διαδικασία ή συνδυασμό αυτών. Η πιο χαρακτηριστική ιδιότητα των fractals είναι ότι είναι γενικά περίπλοκα ως προς τη μορφή τους, δηλαδή εμφανίζουν ανωμαλίες στη μορφή σε σχέση με τα συμβατικά γεωμετρικά σχήματα. Κατά συνέπεια δεν είναι αντικείμενα τα οποία μπορούν να οριστούν με τη βοήθεια της ευκλείδειας γεωμετρίας. Αυτό υποδεικνύεται από το ότι τα fractals, όπως έχει αναφερθεί παραπάνω, έχουν λεπτομέρειες, οι οποίες όμως γίνονται ορατές μόνο μετά από μεγέθυνσή τους σε κάποια κλίμακα.

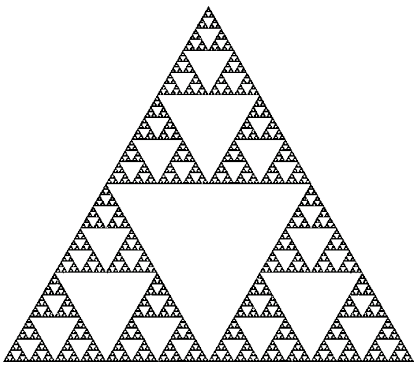
Για να γίνει αντιληπτός αυτός ο διαχωρισμός των fractals σε σχέση με την ευκλείδεια γεωμετρία, αναφέρουμε ότι, αν μεγεθύνουμε κάποιο αντικείμενο το οποίο μπορεί να οριστεί με την ευκλείδεια γεωμετρία, παραδείγματος χάριν την περιφέρεια μιας έλλειψης, αυτή μετά από αλλεπάλληλες μεγεθύνσεις θα εμφανίζεται απλά ως ευθύγραμμο τμήμα. Η συμβατική ιδέα της καμπυλότητας η οποία αντιπροσωπεύει το αντίστροφο της ακτίνας ενός προσεγγίζοντος κύκλου, δεν μπορεί ωφέλιμα να ισχύσει στα fractals επειδή αυτή εξαφανίζεται κατά τη μεγέθυνση. Αντίθετα, σε ένα fractal, θα εμφανίζονται κατόπιν μεγεθύνσεων λεπτομέρειες που δεν ήταν ορατές σε μικρότερη κλίμακα μεγέθυνσης.

Fractals απαντώνται και στη φύση, χωρίς όμως να υπάρχει άπειρη λεπτομέρεια στη μεγέθυνση όπως στα fractals που προκύπτουν από μαθηματικές σχέσεις. Ως παραδείγματα fractals στη φύση, αναφέρονται το σχέδιο των νιφάδων του χιονιού, τα φύλλα των φυτών ή οι διακλαδώσεις των αιμοφόρων αγγείων.

Ο όρος προτάθηκε από τον Μπενουά Μάντελμπροτ (Benoît Mandelbrot) το 1975 και προέρχεται από τη λατινική λέξη fractus, που σημαίνει "σπασμένος", "κατακερματισμένος".

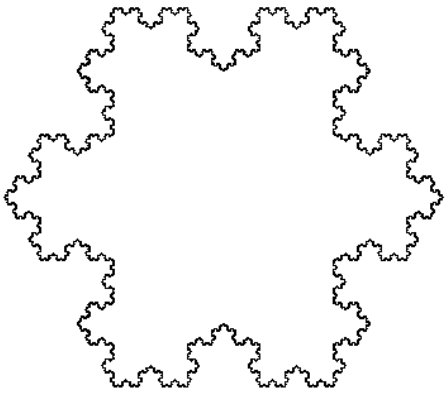
2.1. Fractals δημιουργημένα στον υπολογιστή

2.1.1. Sierpinski Triangle

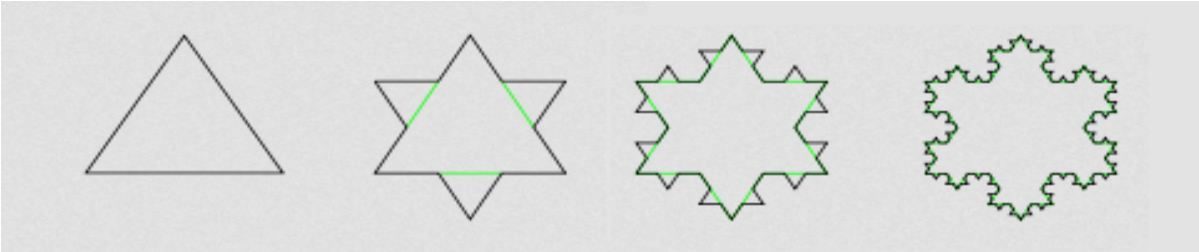


Επαναληπτικά βήματα της κατασκευής του Sierpinski Triangle

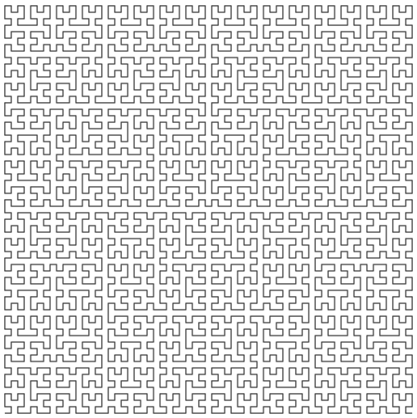
2.1.2. Koch Snowflake



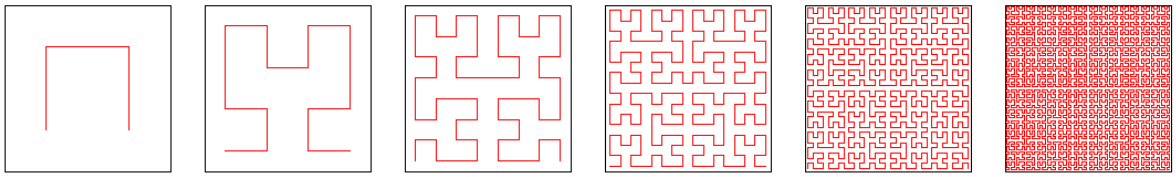
Επαναληπτικά βήματα της κατασκευής του Koch Snowflake



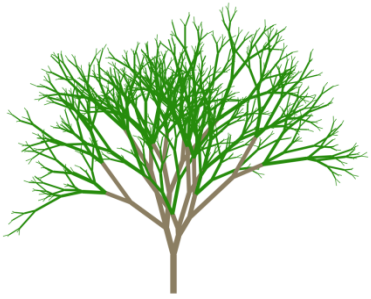
2.1.3. Hilbert Curve



Επαναληπτικά βήματα της κατασκευής της Hilbert Curve

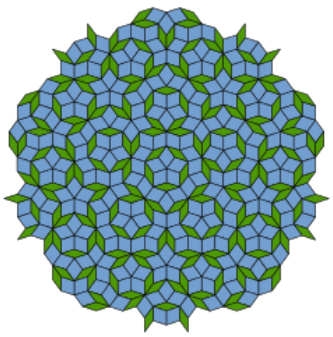


2.1.4. Περισσότερα fractals



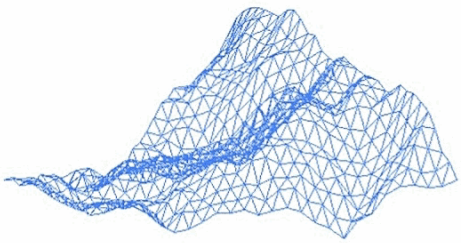
Fractal Tree

Fractal Fern

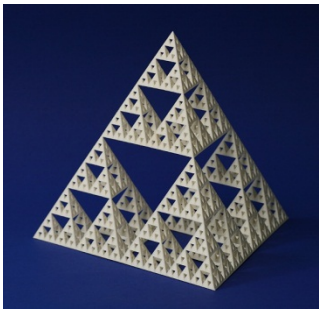


Penrose Tiling

3D Broccoli

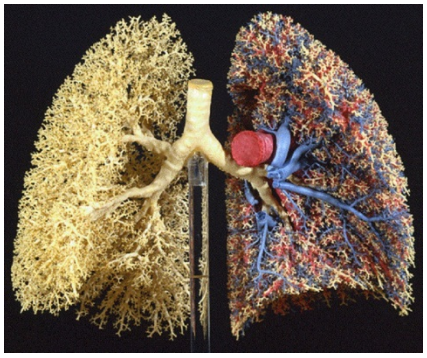


3D Mountain



3D Sierpinski Triangle

2.2. Fractals στον πραγματικό κόσμο



Πνεύμονες

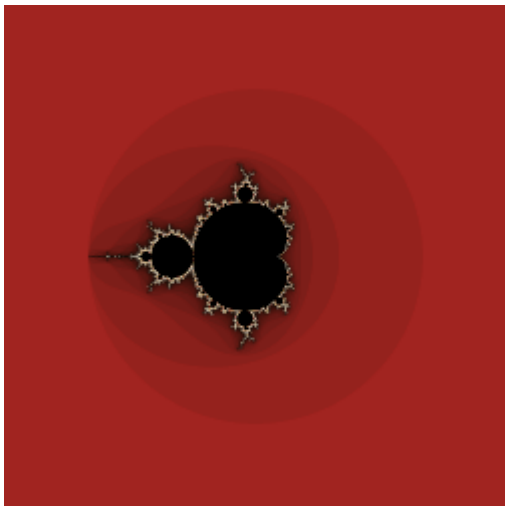
Φύλλο

Δένδρο

Όπως φαινεται ξεκάθαρα και από ολες τις παραπάνω εικόνες, όλα τα αντικείμενα, που χαρακτηρίζονται ως fractals, παράγονται από την επανάληψη κάποιου κανόνα, άλλοτε απλού, άλλοτε πιο σύνθετου.

Η εφαρμογή που αναπτύχθηκε, παράγει fractals που βασίζονται σε επαναληπτικούς κανόνες (συναρτήσεις) μιγαδικών αριθμών, καθόσον η αντιστοίχηση μιας δισδιάστατης εικόνας (i,j συντεταγμένες) στο δισδιάστατο επίπεδο των μιγαδικών αριθμών (real, imaginary συντεταγμένες) είναι αρκετά εύκολη.

Ένα από τα πιο φημισμένα fractals αυτού του τύπου είναι το Mandelbrot set, το οποίο ονομάστηκε έτσι από το μαθηματικό *Benoît Mandelbrot* που το ανακάλυψε. Η ανάλυση της συγκεκριμένης επαναληπτικής συνάρτησης γίνεται σε επόμενο στάδιο.



3. Δημιουργία ενός fractal σε δισδιάστατη εικόνα

Όπως αναφέρθηκε πιο πάνω, η εφαρμογή γράφτηκε σε Java. Αυτό αφ' ενός έκανε εύκολο τον προγραμματισμό του Graphical User Interface (GUI) και αφ' ετέρου εισήγαγε προβλήματα απόδοσης, τα οποία θα αναφερθούν παρακάτω. Ο ψευδοκώδικας που θα παρουσιαστεί προϋποθέτει την ύπαρξη συναρτήσεων που χειρίζονται το γραφικό περιβάλλον της εφαρμογής. Χάριν ευκολίας οι συναρτήσεις αυτές στον ψευδοκώδικα θα έχουν περιγραφικό όνομα για την κατανόηση της λειτουργίας τους.

Για τον σχεδιασμό μιας εικόνας, προφανώς, χρειάζεται με κάποιο τρόπο να είμαστε σε θέση να αποφασίσουμε, τι τιμή χρώματος θα έχει κάθε pixel (Red, Green, Blue form). Αυτό γίνεται με τη βοήθεια μιας συνάρτησης, η οποία παίρνει σαν είσοδο κάποια αριθμητική τιμή και δίνει σαν έξοδο κάποιο χρώμα.

Κατά συνέπεια για να δημιουργήσουμε μια εικόνα διάστασης N x N (υποθέτουμε ότι το width είναι ίσο με το height για τη διευκόλυνση μας), χρειάζεται να κάνουμε το εξής:

Έστω ότι έχουμε μια εικόνα,

```
width = N;
height = N;
image[width][height];
```

```
function drawImage()  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            value = iterativeFunction();
            color = getColor(value);
            paintPixel(image, i, j, color);
        }
```

```
    }

}
```

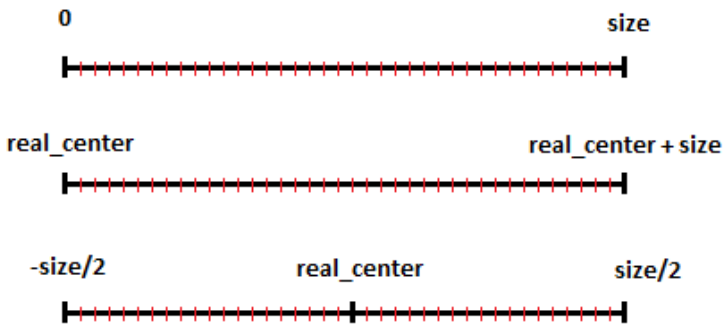
Έστω ότι η iterativeFunction() που αναφέρεται παραπάνω είναι μια οποιαδήποτε επαναληπτική συνάρτηση για την δημιουργία ενός fractal, η οποία αφού εκτελεστεί επιστρέφει μια αριθμητική τιμή. Την τιμή αυτή την περνάμε στη συνάρτηση getColor() και μας επιστρέφει το χρώμα που αναλογεί σε αυτόν τον αριθμό. Αν αυτό το κάνουμε για όλα τα pixels της εικόνας (width x height, N x N) θα έχουμε ολοκληρώσει με επιτυχία την εικόνα μας.

Η εφαρμογή μας όμως θα πρέπει να δουλεύει με μιγαδικούς αριθμούς, δηλαδή η iterativeFunction(), όποια και αν είναι αυτή, θα πρέπει να παίρνει σαν είσοδο ένα μιγαδικό αριθμό. Συνεπώς θα πρέπει να μετατρέψουμε κάθε pixel (i, j) της εικόνας σε ένα μιγαδικό αριθμό (real, imaginary). Αυτό γίνεται με τον ακόλουθο τρόπο:

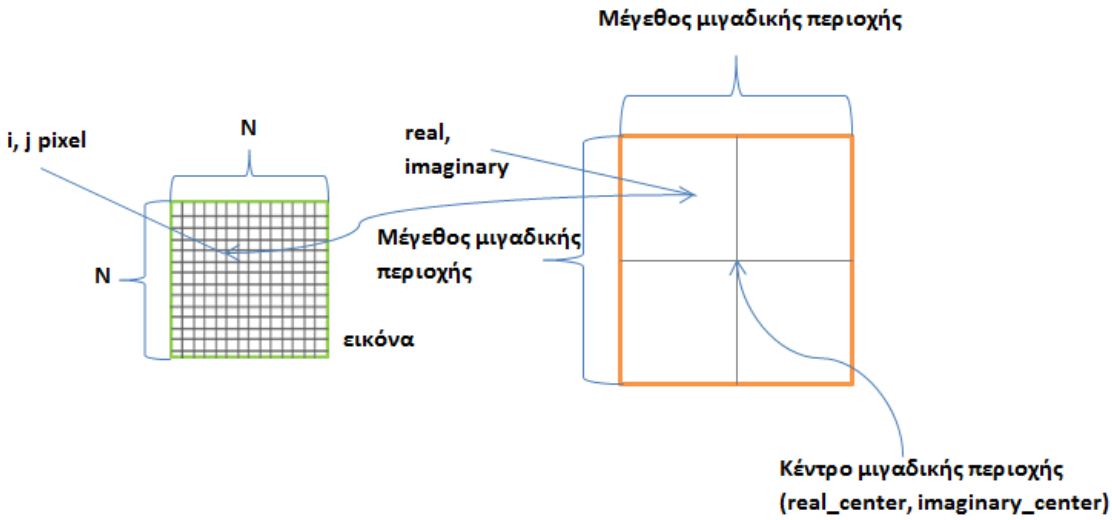
```
function Complex convertPixeltoComplex(int i,  int j) {

    real = real_center - size / 2 + j * (size / width);
    imaginary = imaginary_center - size / 2 + i * (size / height);
    return Complex(real, imaginary);

}
```

Ας πάρουμε για παράδειγμα τον υπολογισμό του real, μιας και ο υπολογισμός του imaginary είναι εντελώς συμμετρικός. Έχουμε width ακέραιες τιμές (pixels), που κάθε μια θέλουμε να την μετατρέψουμε σε μια τιμή που να αντιστοιχεί στο real μέρος ενός μιγαδικού αριθμού. Σημειωτέον ότι μπορούσαμε να κάνουμε 1 προς 1 αντιστοίχηση της ακεραίας τιμής j του pixel σε ένα real, που θα ήταν ακέραιο με την ίδια τιμή. Αυτό όμως δε θα δουλέψει για την δημιουργία ενός fractal, λόγω των ιδιοτήτων των επαναληπτικών συναρτήσεων. Οπότε θα πρέπει να μετατρέψουμε το ακέραιο j σε κάποιο πραγματικό αριθμό. Για αρχή πρέπει να αποφασίσουμε το διάστημα των πραγματικών αριθμών που είμαστε διατιθέμενοι να παραστήσουμε. Έστω ότι αυτό το διάστημα είναι μεγέθους size, άρα θα έχουμε υποδιαιρέσεις μεγέθους $\frac{size}{width}$ του διαστήματος. Ειδικότερα αν το διάστημα αρχίζει από το 0, μας βολεύει το γεγονός ότι και οι ακέραιες τιμές των pixel αρχίζουν από 0 και έτσι μπορούμε να πάρουμε την αντιστοίχιση ενός ακεραίου σε έναν πραγματικό αριθμό χρησιμοποιώντας $j * \frac{size}{width}$. Αν το διάστημα μας δεν αρχίζει από το 0, μπορούμε να προσθέσουμε μια μετατόπιση (είτε θετική είτε αρνητική), οπότε έχουμε $real\_center + j * \frac{size}{width}$. Τέλος, έχουμε συνηθίσει στα γραφήματα να υπάρχει κάποια συμμετρία (0 στο κέντρο, αριστερά αρνητικά, δεξιά θετικά), αυτό όμως δε συμβαίνει στο γραφικό περιβάλλον μας, καθώς το αρχικό pixel είναι πάνω αριστερά στην οθόνη, άρα το ίδιο θα συμβαίνει στην αντιστοίχησή του. Για να λύσουμε αυτό το πρόβλημα, αρκεί να αφαιρέσουμε το μισό διάστημα, $real\_center - \frac{size}{2} + j * \frac{size}{width}$. Έτσι πλέον η αρχή

του διαστήματος έχει μεταφερθεί στη μέση της εικόνας. Για παράδειγμα για size = 6, και real_center = 0 θα έχουμε τιμές που αρχίζουν από το – 3 έως το 3 με υποδιαιρέσεις μεγέθους $\frac{6}{width}$.



Τελική αντιστοίχηση



Συνεπώς ο ψευδοκώδικας για την δημιουργία της εικόνας αλλάζει σε:

```
function drawImage()  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            complex = convertPixeltoComplex(i,  j);
            value = iterativeFunction(complex);
            color = getColor(value);
            paintPixel(image, i, j, color);
        }
    }

}
```

Η getColor() θα αναλυθεί σε επόμενο στάδιο.

4. Fractal συναρτήσεις και Julia sets

Στον παραπάνω ψευδοκώδικα έχει μείνει αδιευκρίνιστο, πως ακριβώς θα λειτουργεί η iterativeFunction(). Αρχικά η iterativeFunction() θα έχει δύο διαφορετικές υλοποιήσεις. Και στις δυο περιπτώσεις ο επαναληπτικός κανόνας θα είναι ο ίδιος και η μόνη διαφοροποίηση θα γίνεται στις αρχικές τιμές. Σε περίπτωση που η iterativeFunction() είναι η fractalFunction() μόνο το pixel χρησιμοποιείται στις αρχικές συνθήκες. Διαφορετικά, αν δηλαδή η iterativeFunction() είναι η juliaFunction() χρησιμοποιείται εκτός του pixel, ακόμα ένας μιγαδικός αριθμός (seed), ο οποίος είναι ίδιος κατά την δημιουργία όλης της εικόνας, δηλαδή η εικόνα κατασκευάζεται βάσει αυτού του αριθμού (seed). Ουσιαστικά η juliaFunction() δημιουργεί τα Julia sets, τα οποία ονομάστηκαν έτσι από τον Gaston Julia.

Ο χρήστης, χρησιμοποιώντας την επιλογή Julia έχει τη δυνατότητα να παρατηρήσει μια προεπισκόπηση των Julia sets της εικόνας, απλά μετακινώντας το δείκτη του ποντικιού στην εικόνα. Φυσικά δύναται να επιλέξει κάποιο μιγαδικό αριθμό ως seed κάνοντας αριστερό mouse click, οπότε θα ζωγραφιστεί το συγκεκριμένο Julia set με μεγαλύτερη ακρίβεια. Το Julia set seed δύναται επίσης να επιλεχθεί μέσω της επιλογής Go To, όπου ο χρήστης επιλέγει συγκεκριμένα το μιγαδικό αριθμό που επιθυμεί να χρησιμοποιήσει σαν seed.

Η επιλογή του seed μέσω του ποντικιού γίνεται από τον παρακάτω ψευδοκώδικα:

```
function  Complex chooseJuliaSeed(int x,  int y) {

    seed_real = real_center - size / 2 + y * (size / width);
    seed_imaginary = imaginary_center - size / 2 + x * (size /
height);
    return Complex(seed_real, seed_imaginary);

}
```

Όπου x, y είναι οι συντεταγμένες του ποντικιού.

Σκόπιμο είναι σε αυτό το σημείο να γίνει μια επανάληψη στις πράξεις μεταξύ μιγαδικών αριθμών.

- Πρόσθεση: $(\boldsymbol{a} + \boldsymbol{bi}) + (\boldsymbol{c} + \boldsymbol{di}) = (\boldsymbol{a} + \boldsymbol{c}) + (\boldsymbol{b} + \boldsymbol{d})\boldsymbol{i}$
- Αφαίρεση: $(\boldsymbol{a} + \boldsymbol{bi}) - (\boldsymbol{c} + \boldsymbol{di}) = (\boldsymbol{a} - \boldsymbol{c}) + (\boldsymbol{b} - \boldsymbol{d})\boldsymbol{i}$
- Πολλαπλασιασμός: $(\boldsymbol{a} + \boldsymbol{bi}) * (\boldsymbol{c} + \boldsymbol{di}) = (\boldsymbol{ac} - \boldsymbol{bd}) + (\boldsymbol{ad} + \boldsymbol{bc})\boldsymbol{i}$
- Διαίρεση: $\frac{(\boldsymbol{a} + \boldsymbol{bi})}{(\boldsymbol{c} + \boldsymbol{di})} = (\frac{\boldsymbol{ac} + \boldsymbol{bd}}{\boldsymbol{c^2} + \boldsymbol{d^2}}) + (\frac{\boldsymbol{bc} - \boldsymbol{ad}}{\boldsymbol{c^2} + \boldsymbol{d^2}})\boldsymbol{i}$

- Συζυγής μιγαδικός αριθμός: Αν $\boldsymbol{Z = a + bi}$, $\boldsymbol{\bar{Z} = a - bi}$
- Ευκλείδεια νόρμα (μέτρο): Αν $\boldsymbol{Z = a + bi}$, $\boldsymbol{|Z| = \sqrt{a^2 + b^2}}$
- Ύψωση σε δύναμη: Αν $\boldsymbol{Z = a + bi}$, για $\boldsymbol{Z^2}$, αντίστοιχα για ακέραια n.
    1. $\boldsymbol{Z * Z}$ (Πολλαπλασιασμοί)
    2. $\boldsymbol{(a + bi)^2} = \mathbf{(a^2 - b^2) + 2abi}$ (Binomial)

Σε όλες τις επαναληπτικές συναρτήσεις, εκτός του επαναληπτικού κανόνα, υπάρχει κάποια συνθήκη τερματισμού (της επανάληψης) και αυτή η συνθήκη, μιας και χρησιμοποιούμε μιγαδικούς αριθμούς είναι η ευκλείδεια νόρμα (μέτρο) του μιγαδικού αριθμού.

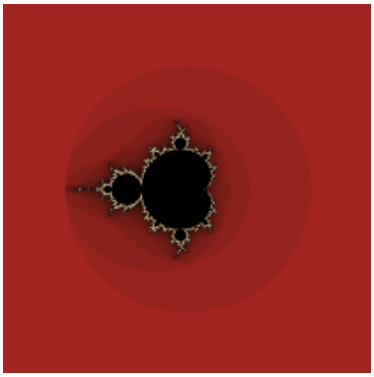
- $\boldsymbol{|Z_n| \geq bailout}$ (Κριτήριο άνω φράγματος)
- $\boldsymbol{|Z_n - Z_{n-1}| \leq error}$ (Κριτήριο σύγκλισης, Root Finding Methods)
- Κάποιος συνδυασμός των δύο παραπάνω (Magnet functions)

Όπου το bailout δίνεται από το χρήστη της εφαρμογής (συνήθως bailout = 2), ενώ το error έχει κάποια πολύ μικρή τιμή (συνήθως error = 1e-8) με σκοπό τη σύγκλιση σε κάποια ρίζα.

Ακόμα μια συνθήκη τερματισμού των επαναληπτικών συναρτήσεων είναι ένας μέγιστος αριθμός επαναλήψεων (max_iterations στον ψευδοκώδικα), ο οποίος προσδιορίζεται από το χρήστη.

Ακολουθεί η παρουσίαση του ψευδοκώδικα για κάθε συνάρτηση της εφαρμογής, περιλαμβάνοντας τις συναρτήσεις fractalFunction() και juliaFunction() για κάθε μια αντίστοιχα (εκτός των Root Finding Methods και Sierpinski Gasket που δεν έχουν Julia sets).

4.1. Mandelbrot



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^2 + c}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
    }

    return max_iterations;
}
```

Όπως παρατηρούμε, σε αυτό το σημείο οι συναρτήσεις fractalFunction() και juliaFunction() μοιάζουν αρκετά. Επίσης οι συναρτήσεις χρησιμοποιούν ακόμα μια συνάρτηση colorAlgorithm(), που για την ώρα η υλοποίηση της καθώς και οι παράμετροι της δεν έχουν σημασία. Αρχικά θα χρησιμοποιήσουμε στην colorAlgorithm() μόνο τον αριθμό των επαναλήψεων, στον οποίο τερμάτισε η συνάρτηση. Η αναλυση της colorAlgorithm() θα γινει σε επομενο σταδιο.

4.2. Mandelbrot Cubed

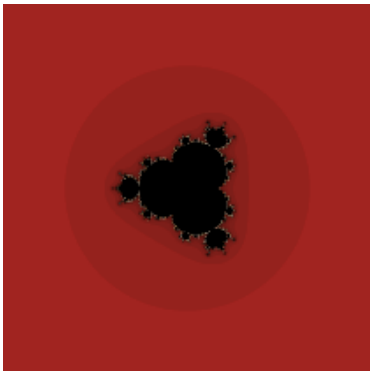
Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^3 + c}$

Συνθήκη τερματισμού της επανάληψης: $\boldsymbol{|z| \geq bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^3 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^3 + c;
    }

    return max_iterations;
}
```

4.3. Mandelbrot Fourth



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^4 + c}$

Συνθήκη τερματισμού της επανάληψης: $|z| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^4 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^4 + c;
    }

    return max_iterations;
}
```

4.4. Mandelbrot Fifth

Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^5 + c}$

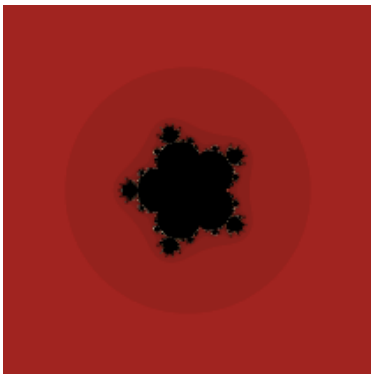
Συνθήκη τερματισμού της επανάληψης: $|z| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
```

```
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^5 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^5 + c;
    }

    return max_iterations;
}
```

4.5. Mandelbrot Sixth



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^6 + c}$

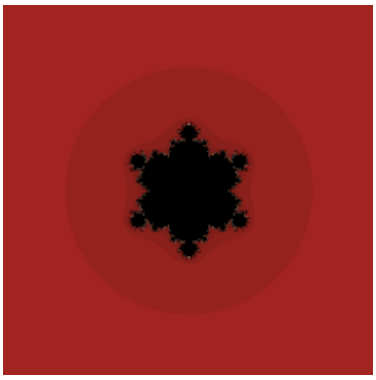
Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
```

```
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^6 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^6 + c;
    }

    return max_iterations;
}
```

4.6. Mandelbrot Seventh



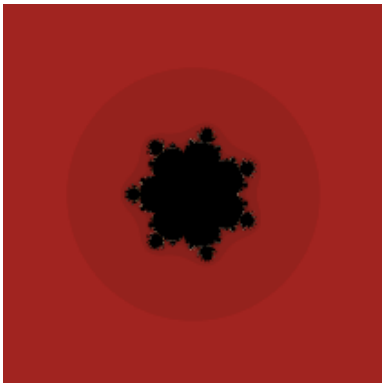
Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^7 + c}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^7 + c;
```

```
    }

    return max_iterations;
}
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^7 + c;
    }

    return max_iterations;
}
```

4.7. Mandelbrot Eighth



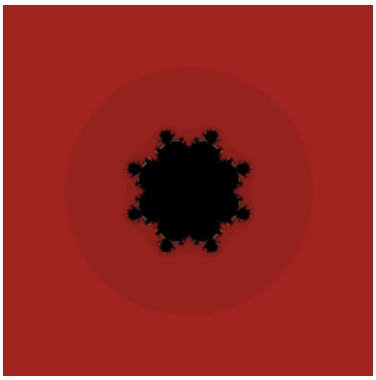
Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^8 + c}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^8 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^8 + c;
    }

    return max_iterations;
}
```

4.8. Mandelbrot Ninth



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^9 + c}$

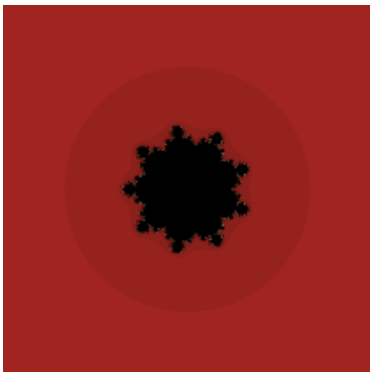
Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^9 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;
```

```
    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^9 + c;
    }

    return max_iterations;
}
```

4.9. Mandelbrot Tenth



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^{10} + c}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^10 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
```

```
        }
        z = z^10 + c;
    }

    return max_iterations;
}
```

4.10. Mandelbrot Nth

Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^n + c}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^n + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^n + c;
    }

    return max_iterations;
}
```

Όπως παρατηρούμε σε αυτό το σημείο το z^n, όταν το n είναι πραγματικός αριθμός, το οποίο δίνεται από το χρήστη, δε μπορεί να υπολογιστεί με το συμβατικό τρόπο. Οπότε θα πρέπει να μετατρέπουμε σε κάθε βήμα το μιγαδικό αριθμό σε πολικές συντεταγμένες (μέτρο και φάση) , ώστε να γίνει εύκολα η ύψωση στη δύναμη, και μετά πάλι στη μορφή του μιγαδικού αριθμού. Χρησιμοποιούμε τις συναρτήσεις της math library (pow, atan2, cos, sin), οπότε έχουμε $norm\_n = pow(|z|, n)$ και $phase\_n = n * atan2(Im(z), Re(z))$. Στο τέλος μετατρέπουμε ξανά σε μιγαδικό αριθμό, $z = norm\_n * cos(phase\_n) +$

$norm\_n * sin(phase\_n)i$. Φυσικά αυτές οι επιπλέον μετατροπές, καθώς και η χρήση τριγωνομετρικών συναρτήσεων κάνουν τους υπολογισμούς υπερβολικά ακριβούς. Επομένως ο υπολογισμός ολόκληρης της εικόνας είναι πολύ χρονοβόρος.

4.11. Mandelbrot Polynomial

Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = p(z) + c}$

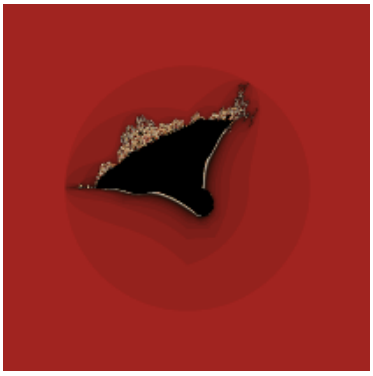
Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = p(z) + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = p(z) + c;
    }

    return max_iterations;
}
```

Όπου p(z) είναι ένα πολυώνυμο, μέχρι και $10^{ου}$ βαθμού, το οποίο δίνεται από το χρήστη. Σε κάθε βήμα θα πρέπει να υπολογιστεί η τιμή του πολυωνύμου για το τρέχον z.

4.12. Burning Ship



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = (|a| + |b|i)^2 + c}$

Συνθήκη τερματισμού της επανάληψης: $\boldsymbol{|z| \geq bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = Complex(abs(z.real), abs(z.imaginary));
        z = z^2 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = Complex(abs(z.real), abs(z.imaginary));
        z = z^2 + c;
    }

    return max_iterations;
}
```

Η επιλογή burning ship μπορεί να εφαρμοστεί μόνο στις παραπάνω συναρτήσεις (Mandelbrot, Mandelbrot Cubed, …, Mandelbrot Polynomial), ενώ στις υπόλοιπες δεν έχει κανένα αποτέλεσμα. Συγκεκριμένα θα δείξουμε την αλλαγή που πρέπει να υποστεί η

συνάρτηση Mandelbrot, καθώς όλες οι άλλες πρέπει να υποστούν το ίδιο. Σε κάθε βήμα πριν τον υπολογισμό του $z = z^2 + c$, θα πρέπει να αλλάζουμε το μιγαδικό, ώστε να είναι πάντα θετικός, χρησιμοποιώντας την απόλυτη τιμή του real και του imaginary.

4.13. Mandelbar

Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = \bar{z}^2 + c}$

Συνθήκη τερματισμού της επανάληψης: $\boldsymbol{|z| \geq bailout}$

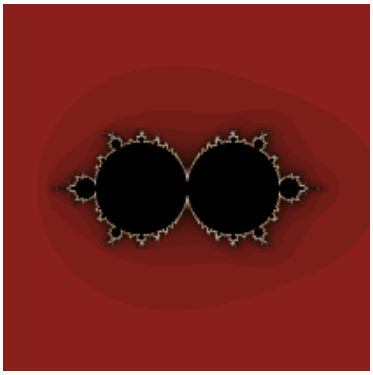
```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = conjugate(z);
        z = z^2 + c;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = conjugate(z);
        z = z^2 + c;
    }

    return max_iterations;
```

```
}
```

Όπου η συνάρτηση conjugate() επιστρέφει το συζυγή μιγαδικό αριθμό.

4.14. Lambda



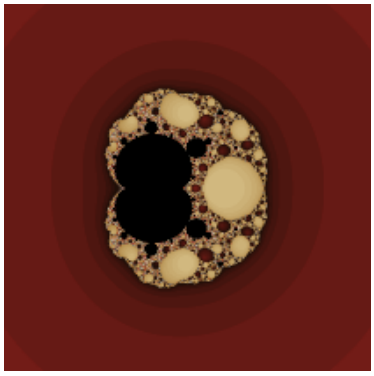
Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = cz(1 - z)}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = Complex(0.5, 0);
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = c * z * (1 - z);
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = c * z * (1 - z);
    }

    return max_iterations;
}
```

4.15. Magnet 1



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z} = \left(\frac{\boldsymbol{z^2+c-1}}{\boldsymbol{2z+c-2}}\right)^{\boldsymbol{2}}$

Συνθήκες τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$ ή $|\boldsymbol{z-1}| \leq \boldsymbol{error}$

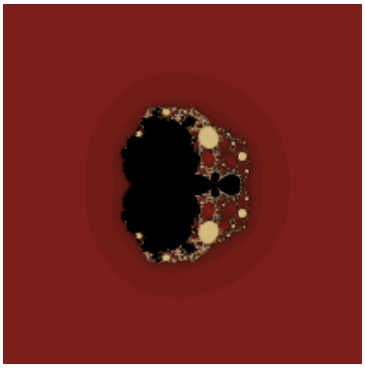
```
function double fractalFunction(Complex pixel) {
    z = Complex(0, 0);
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout || |z - 1| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        z = ((z^2 + c - 1) / (2 * z + c - 2))^2;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout || |z - 1| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        z = ((z^2 + c - 1) / (2 * z + c - 2))^2;
    }

    return max_iterations;
}
```

Οι magnet functions είναι οι μοναδικές συναρτήσεις που συνδυάζουν δύο συνθήκες τερματισμού, οπότε αρκεί να ισχύει η μία από τις δύο. Η δεύτερη συνθήκη μοιάζει με την $|Z_n - Z_{n-1}| \leq error$, η οποία είναι πιο γενικός τύπος για τη σύγκλιση. Όταν γνωρίζουμε

απευθείας την τιμή της ρίζας, μπορούμε να τη χρησιμοποιήσουμε απευθείας στον τύπο. Σε αυτή την περίπτωση η συνάρτηση συγκλίνει στο 1, οπότε έχουμε $|Z_n - 1| \leq error$.

4.16. Magnet 2



Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z} = \left(\frac{z^3+3(c-1)z+(c-1)(c-2)}{3z^2+3(c-2)+(c-1)(c-2)+1}\right)^2$

Συνθήκες τερματισμού της επανάληψης: $|z| \geq bailout$ ή $|z-1| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = Complex(0, 0);
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout || |z - 1| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        z = ((z^3 + 3 * (c - 1) * z + (c - 1) * (c - 2)) / (3 * z^2 +
3 * (c - 2) * z + (c - 1) * (c - 2) + 1))^2;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout || |z - 1| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        z = ((z^3 + 3 * (c - 1) * z + (c - 1) * (c - 2)) / (3 * z^2 +
3 * (c - 2) * z + (c - 1) * (c - 2) + 1))^2;
    }

    return max_iterations;
}
```

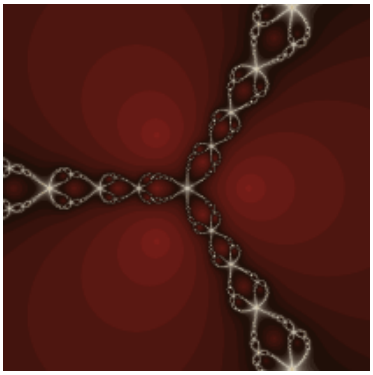
4.17. Root Finding Methods

Οι εν λόγω προσπαθούν, με τη χρήση επαναληπτικών τύπων, να συγκλίνουν στις ρίζες μιας συνάρτησης μέχρι να προκύψει κάποιο αποδεκτό σφάλμα ακρίβειας (error). Στην εφαρμογή, όλες οι root finding methods χρησιμοποιούν πολυωνυμικές συναρτήσεις, λόγω της ευκολίας υπολογισμού τους. Οι root finding methods περιλαμβάνουν τους τύπους του Newton, Halley, Schroder και Householder.

4.17.1. Newton

Χρησιμοποιείται ο επαναληπτικός τύπος Newton–Raphson, $X_{n+1} = X_n - \frac{f(Xn)}{f'(Xn)}$. Φυσικά ο τύπος ισχύει και για μιγαδικές συναρτήσεις / αριθμούς.

4.17.1.1. Newton 3



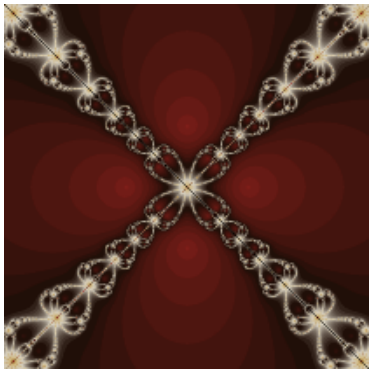
$f(z) = z^3 - 1,\ f'(z) = 3z^2$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^3 - 1) / (3 * z^2));
    }

    return max_iterations;
}
```

Όλες οι Root Finding Methods δεν έχουν juliaFunction(), διότι όπως παρατηρούμε έχουν μόνο ένα βαθμό ελευθερίας, οπότε δε μπορούμε να χρησιμοποιήσουμε μια σταθερή τιμή σαν seed. Σε όλες τις root finding methods χρησιμοποιείται το κριτήριο σύγκλισης ως συνθήκη τερματισμού της επανάληψης.

4.17.1.2. Newton 4

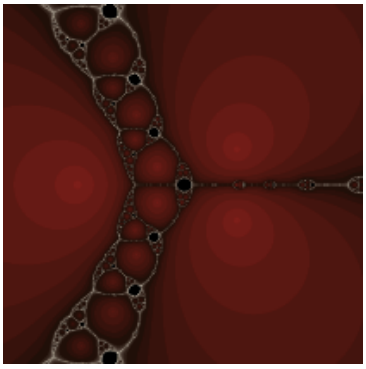


$$f(z) = z^4 - 1,\ f'(z) = 4z^3$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^4 - 1) / (4 * z^3));
    }

    return max_iterations;
}
```

4.17.1.3. Newton Generalized 3

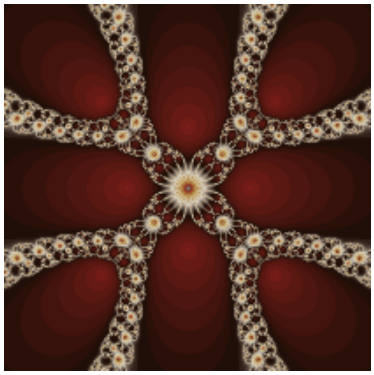


$$f(z) = z^3 - 2z + 2,\ f'(z) = 3z^2 - 2$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^3 - 2 * z + 2) / (3 * z^2 - 2));
    }

    return max_iterations;
}
```

4.17.1.4. Newton Generalized 8



$$f(z) = z^8 + 15z - 16,\ f'(z) = 8z^7 + 60z^3$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^8 + 15 * z^4 - 16) / (8 * z ^ 7 + 60 * z ^ 3));
    }

    return max_iterations;
}
```

4.17.1.5. *Newton Polynomial*

$p(z)$, $p'(z)$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

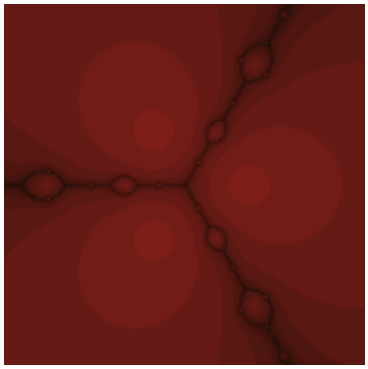
```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - p(z) / p'(z);
    }

    return max_iterations;
}
```

Όπου p(z) είναι ένα πολυώνυμο, μέχρι και 10ου βαθμού, το οποίο δίνεται από το χρήστη. Σε κάθε βήμα θα πρέπει να υπολογιστεί η τιμή του πολυωνύμου και η παράγωγος του για το τρέχον z.

4.17.2. Halley

Χρησιμοποιείται ο επαναληπτικός τύπος Halley, $X_{n+1} = X_n - \frac{2f(Xn) * f'(Xn)}{2f'(Xn)^2 - f(Xn) * f''(Xn)}$.
Φυσικά ο τύπος ισχύει και για μιγαδικές συναρτήσεις / αριθμούς.

4.17.2.1. Halley 3

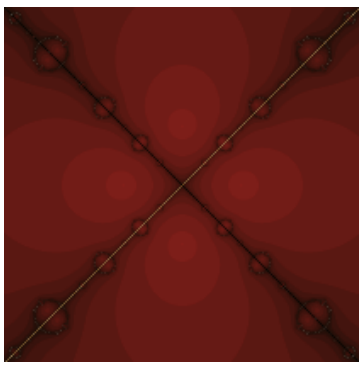


$$f(z) = z^3 - 1,\ f'(z) = 3z^2,\ f''(z) = 6z$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - (2 * (z^3 - 1) * (3 * z^2)) / (2 * (3 * z^2)^2 - (z^3
- 1) * (6 * z));
    }

    return max_iterations;
}
```

4.17.2.2. Halley 4

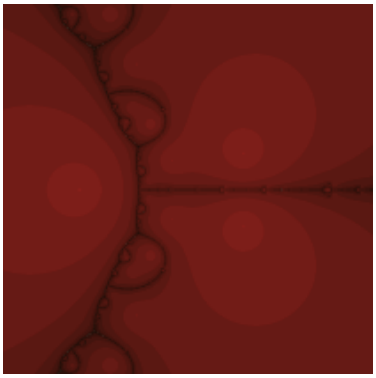


$f(z) = z^4 - 1,\ f'(z) = 4z^3,\ f''(z) = 12z^2$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - (2 * (z^4 - 1) * (4 * z^3)) / (2 * (4 * z^3)^2 - (z^4
- 1) * (12 * z^2));
    }

    return max_iterations;
}
```

4.17.2.3. *Halley Generalized 3*

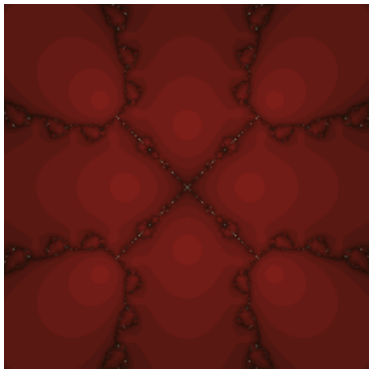


$f(z) = z^3 - 2z + 2,\ f'(z) = 3z^2 - 2,\ f''(z) = 6z$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - (2 * (z^3 - 2 * z + 2) * (3 * z^2 - 2)) / (2 * (3 *
z^2 - 2)^2 - (z^3 - 2 * z + 2) * (6 * z));
    }

    return max_iterations;
}
```

4.17.2.4. Halley Generalized 8



$$f(z) = z^8 + 15z^4 - 16,\ f'(z) = 8z^7 + 60z^3,\ f''(z) = 56z^6 + 180z^2$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - (2 * (z^8 + 15 * z^4 - 16) * (8 * z ^ 7 + 60 * z ^
3)) / (2 * (8 * z ^ 7 + 60 * z ^ 3)^2 - (z^8 + 15 * z^4 - 16) * (56 *
z^6 + 180 * z^2));
    }

    return max_iterations;
}
```

4.17.2.5. Halley Polynomial

$$p(z),\ p'(z),\ p''(z)$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - (2 * p(z) * p'(z)) / (2 * p'(z)^2 - p(z) * p''(z));
```

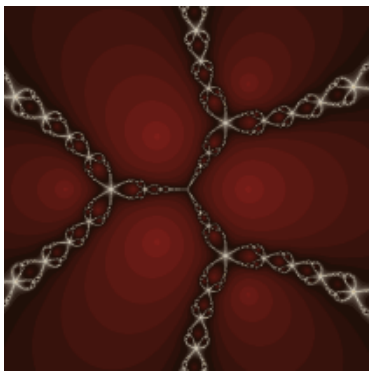
```
    }

    return max_iterations;
}
```

Όπου p(z) είναι ένα πολυώνυμο, μέχρι και 10ου βαθμού, το οποίο δίνεται από το χρήστη. Σε κάθε βήμα θα πρέπει να υπολογιστεί η τιμή του πολυωνύμου καθώς και 1η, 2η παράγωγος για το τρέχον z.

4.17.3. Schroder

Χρησιμοποιείται ο επαναληπτικός τύπος Schroder, $X_{n+1} = X_n - \frac{f(Xn) * f'(Xn)}{f'(Xn)^2 - f(Xn) * f''(Xn)}$. Φυσικά ο τύπος ισχύει και για μιγαδικές συναρτήσεις / αριθμούς.

4.17.3.1. Schroder 3



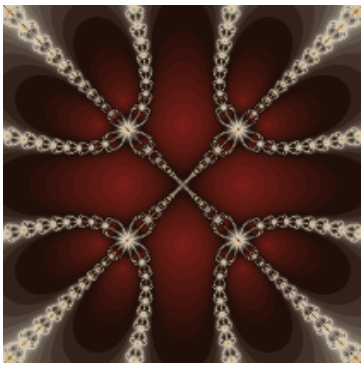
$f(z) = z^3 - 1,\ f'(z) = 3z^2,\ f''(z) = 6z$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^3 - 1) * (3 * z^2)) / ((3 * z^2)^2 - (z^3 - 1) *
(6 * z));
    }
```

```
    return max_iterations;
}
```

4.17.3.2. Schroder 4

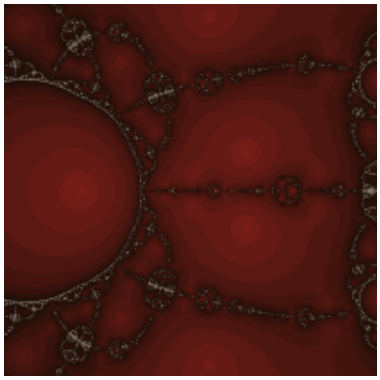


$$f(z) = z^4 - 1,\ f'(z) = 4z^3,\ f''(z) = 12z^2$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ( (z^4 - 1) * (4 * z^3)) / ( (4 * z^3)^2 - (z^4 - 1)
* (12 * z^2));
    }

    return max_iterations;
}
```

4.17.3.3. Schroder Generalized 3

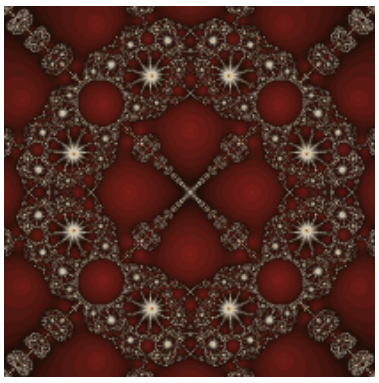


$$f(z) = z^3 - 2z + 2,\ f'(z) = 3z^2 - 2,\ f''(z) = 6z$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^3 - 2 * z + 2) * (3 * z^2 - 2)) / ((3 * z^2 -
2)^2 - (z^3 - 2 * z + 2) * (6 * z));
    }

    return max_iterations;
}
```

4.17.3.4. Schroder Generalized 8



$$f(z) = z^8 + 15z^4 - 16,\ f'(z) = 8z^7 + 60z^3,\ f''(z) = 56z^6 + 180z^2$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^8 + 15 * z^4 - 16) * (8 * z ^ 7 + 60 * z ^ 3)) /
((8 * z ^ 7 + 60 * z ^ 3)^2 - (z^8 + 15 * z^4 - 16) * (56 * z^6 + 180
* z^2));
    }

    return max_iterations;
}
```

4.17.3.5. *Schroder Polynomial*

$p(z),\ p'(z),\ p''(z)$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

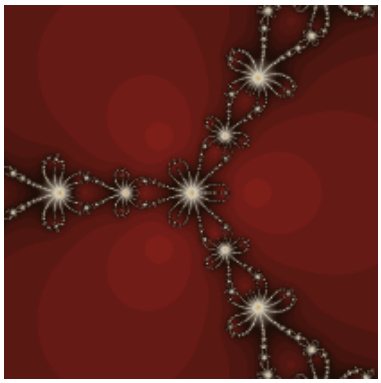
```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - (p(z) * p'(z)) / (p'(z)^2 - p(z) * p''(z));
    }

    return max_iterations;
}
```

Όπου p(z) είναι ένα πολυώνυμο, μέχρι και 10^{ου} βαθμού, το οποίο δίνεται από το χρήστη. Σε κάθε βήμα θα πρέπει να υπολογιστεί η τιμή του πολυωνύμου καθώς και 1^{η}, 2^{η} παράγωγος για το τρέχον z.

4.17.4. Householder

Χρησιμοποιείται ο επαναληπτικός τύπος Householder, $\boldsymbol{X_{n+1} = X_n - \frac{f(Xn) * [2f'(Xn)^2 + f(Xn) * f''(Xn)]}{2f'(Xn)^3}}$. Φυσικά ο τύπος ισχύει και για μιγαδικές συναρτήσεις / αριθμούς.

4.17.4.1. Householder 3

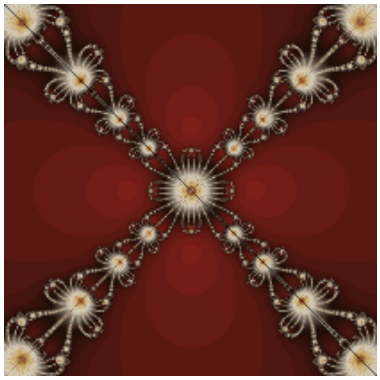


$\boldsymbol{f(z) = z^3 - 1,\ f'(z) = 3z^2,\ f''(z) = 6z}$

Συνθήκη τερματισμού της επανάληψης: $\boldsymbol{|z_i - z_{i-1}| \leq error}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^3 - 1) * (2 * (3 * z^2)^2 + (z^3 - 1) * (6 * z))
/ (2 * (3 * z^2)^3);
    }

    return max_iterations;
}
```

4.17.4.2. Householder 4

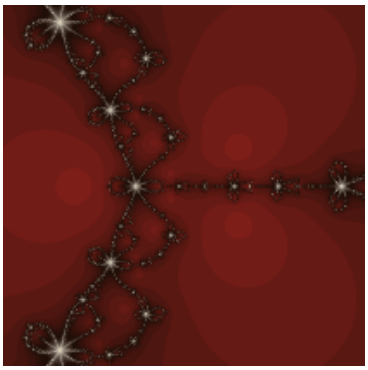


$$f(z) = z^4 - 1,\ f'(z) = 4z^3,\ f''(z) = 12z^2$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^4 - 1) * (2 * (4 * z^3)^2 + (z^4 - 1) * (12 *
z^2)) / (2 * (4 * z^3)^3);
    }

    return max_iterations;
}
```

4.17.4.3. Householder Generalized 3

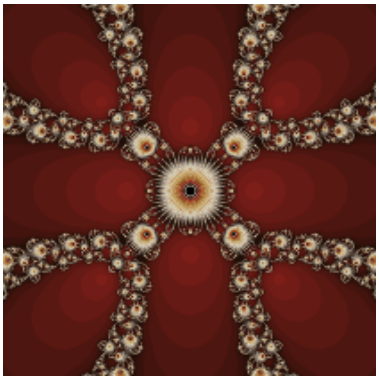


$$f(z) = z^3 - 2z + 2,\ f'(z) = 3z^2 - 2,\ f''(z) = 6z$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^3 - 2 * z + 2) * (2 * (3 * z^2 - 2)^2 + (z^3 - 2
* z + 2) * (6 * z)) / (2 * (3 * z^2 - 2)^3);
    }

    return max_iterations;
}
```

4.17.4.4. Householder Generalized 8



$$f(z) = z^8 + 15z^4 - 16,\ f'(z) = 8z^7 + 60z^3,\ f''(z) = 56z^6 + 180z^2$$

Συνθήκη τερματισμού της επανάληψης: $|z_i - z_{i-1}| \leq error$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - ((z^8 + 15 * z^4 - 16) * (2 * (8 * z ^ 7 + 60 * z ^
3)^2 + (z^8 + 15 * z^4 - 16) * (56 * z^6 + 180 * z^2)) / (2 * (8 * z
^ 7 + 60 * z ^ 3)^3);
    }

    return max_iterations;
}
```

4.17.4.5. Householder Polynomial

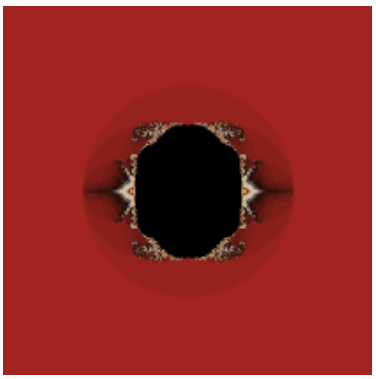
$\boldsymbol{p(z)},\ \boldsymbol{p'(z)},\ \boldsymbol{p''(z)}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z_i - z_{i-1}}| \leq \boldsymbol{error}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(iterations != 0  && |z - zold| <= error) {
            return colorAlgorithm(iterations,…,);
        }
        zold = z;
        z = z - (p(z) * (2 * p'(z)^2 + p(z) * p''(z)) / (2 * p'(z)^3);
    }

    return max_iterations;
}
```

Όπου p(z) είναι ένα πολυώνυμο, μέχρι και 10^{ου} βαθμού, το οποίο δίνεται από το χρήστη. Σε κάθε βήμα θα πρέπει να υπολογιστεί η τιμή του πολυωνύμου καθώς και 1^{η}, 2^{η} παράγωγος για το τρέχον z.

4.18. Barnsley 1



Επαναληπτικός κανόνας της συνάρτησης:

$\boldsymbol{if\ Re(z) \geq 0\ then\ z = (z-1)c\ else\ z = (z+1)c}$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;
```

```
    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }

        if(Re(z) >= 0) {
            z = (z - 1) * c;
        }
        else {
            z = (z + 1) * c;
        }
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }

        if(Re(z) >= 0) {
            z = (z - 1) * c;
        }
        else {
            z = (z + 1) * c;
        }
    }

    return max_iterations;
}
```

4.19. Barnsley 2

Επαναληπτικός κανόνας της συνάρτησης:

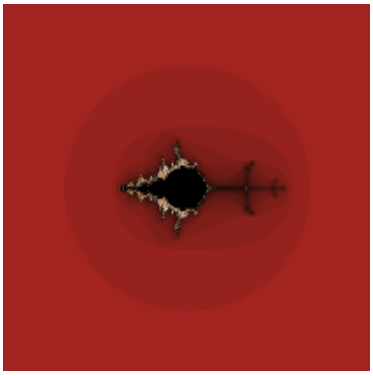
$$\boldsymbol{if}\ (\boldsymbol{Re}(\boldsymbol{z})\boldsymbol{Im}(\boldsymbol{c}) + \boldsymbol{Re}(\boldsymbol{c})\boldsymbol{Im}(\boldsymbol{z})) \geq \boldsymbol{0}\ \boldsymbol{then}\ \boldsymbol{z} = (\boldsymbol{z}+\boldsymbol{1})\boldsymbol{c}\ \boldsymbol{else}\ \boldsymbol{z} = (\boldsymbol{z}-\boldsymbol{1})\boldsymbol{c}$$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }

        if(Re(z) * Im(c) + Re(c) * Im(z) >= 0) {
            z = (z + 1) * c;
        }
        else {
            z = (z - 1) * c;
        }
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }

        if(Re(z) * Im(c) + Re(c) * Im(z) >= 0) {
            z = (z + 1) * c;
        }
        else {
            z = (z - 1) * c;
        }
    }

    return max_iterations;
}
```

4.20. Barnsley 3



Επαναληπτικός κανόνας της συνάρτησης:

$$\boldsymbol{if\ Re(z) > 0\ then\ z\ = z^2 - 1\ else\ z\ = z^2 + (Re(c)Re(z) + Im(c)Re(z)i) - 1}$$

Συνθήκη τερματισμού της επανάληψης: $|\boldsymbol{z}| \geq \boldsymbol{bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }

        if(Re(z) > 0) {
            z = z^2 - 1;
        }
        else {
            z = z^2 + Complex(Re(c) * Re(z), Im(c) * Re(z)) - 1;
        }
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }

        if(Re(z) > 0) {
            z = z^2 - 1;
        }
        else {
            z = z^2 + Complex(Re(c) * Re(z), Im(c) * Re(z)) - 1;
```

```
        }
    }

    return max_iterations;
}
```

4.21. Spider

Επαναληπτικός κανόνας της συνάρτησης: $\boldsymbol{z = z^2 + c,\ c = \frac{c}{2} + z}$

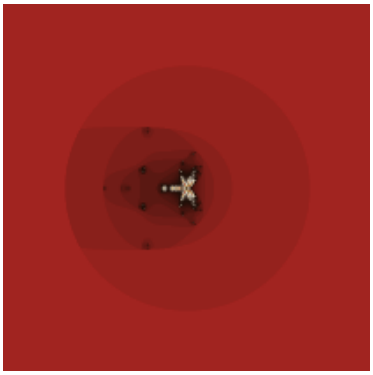
Συνθήκη τερματισμού της επανάληψης: $\boldsymbol{|z| \geq bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
        c = c / 2 + z;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
        c = c / 2 + z;
    }
```

```
    return max_iterations;
}
```

4.22. Manowar



Επαναληπτικός κανόνας της συνάρτησης: $t = z^2 + z_1 + c,\ z_1 = z,\ z = t$

Συνθήκη τερματισμού της επανάληψης: $|z| \geq bailout$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    z1 = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        t = z^2 + z1 + c;
        z1 = z;
        z = t;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    z1 = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        t = z^2 + z1 + c;
        z1 = z;
```

```
        z = t;
    }

    return max_iterations;
}
```

4.23. Phoenix

Επαναληπτικός κανόνας της συνάρτησης:

$$t = z^2 + (Im(c)Re(s) + Re(c) + Im(c)Im(s)i),\ s = z,\ z = t$$

Συνθήκη τερματισμού της επανάληψης: $|z| \geq bailout$

```
function double fractalFunction(Complex pixel) {
    z = pixel;
    c = pixel;
    s = Complex(0,0);

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        t = z^2 + Complex(Im(c) * Re(s) + Re(c), Im(c) * Im(s));
        s = z
        z = t;
    }

    return max_iterations;
}
```

```
function double juliaFunction(Complex pixel) {
    z = pixel;
    c = seed;
    s = Complex(0,0);

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
```

```
            return colorAlgorithm(iterations,…,);
        }
        t = z^2 + Complex(Im(c) * Re(s) + Re(c), Im(c) * Im(s));
        s = z
        z = t;
    }

    return max_iterations;
}
```

4.24. Sierpinski Gasket

Επαναληπτικός κανόνας της συνάρτησης:

$$\boldsymbol{if\ Im(z) > 0.5\ then\ z\ = 2Re(z) + (2Im(z) - 1)i}$$

$$\boldsymbol{else\ if\ Re(z) > 0.5\ then\ z = 2Re(z) - 1 + 2Im(z)i}$$

$$\boldsymbol{else\ z\ =\ 2z}$$

Συνθήκη τερματισμού της επανάληψης: $\boldsymbol{|z| \geq bailout}$

```
function double fractalFunction(Complex pixel) {
    z = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }

        if(Im(z) > 0.5) {
            z = Complex(2 * Re(z), 2 * Im(z) - 1);
        }
        else if(Re(z) > 0.5) {
            z = Complex(2 * Re(z) - 1, 2 * Im(z));
        }
        else {
```

```
                z = 2 * z;
            }
        }

        return max_iterations;
    }
```

Η συνάρτηση Sierpinski Gasket δεν έχει juliaFunction(), διότι όπως παρατηρούμε έχει μόνο ένα βαθμό ελευθερίας, οπότε δε μπορούμε να χρησιμοποιήσουμε μια σταθερή τιμή σαν seed.

Όλες οι προαναφερθείσες συναρτήσεις έχουν μεγάλο βαθμό ομοιότητας, κυρίως στο επαναληπτικό κομμάτι. Η κληρονομικότητα και ο πολυμορφισμός, που παρέχει η Java, μας βοηθά αρκετά, ώστε να μπορούμε να δημιουργήσουμε μια βασική επαναληπτική μέθοδο, η οποία θα διαφοροποιείται μόνο στο κομμάτι του κανόνα κάθε συνάρτησης. Αν οι διαφορές είναι πιο πολλές, μπορούμε απλά να κάνουμε override τη μέθοδο της αντίστοιχης συνάρτησης.

Επειδή τα Julia sets μπορούν να πάρουν άπειρες τιμές σαν seed, προφανώς προκύπτουν και άπειρες εικόνες, οπότε σε αυτό το σημείο δε θα παρουσιαστεί καμία εικόνα τους.

5. Επιλογές χρωμάτων

5.1. Αλγόριθμοι χρωματισμού

Μέχρι αυτό το σημείο έχουμε καθορίσει τις επαναληπτικές συναρτήσεις, αλλά δεν έχει καθοριστεί ακόμα, τί είδους τιμές ακριβώς επιστρέφουν (σε προηγούμενο βήμα είπαμε ότι χρησιμοποιούμε τον αριθμό των επαναλήψεων).

Όπως αναφέραμε, υπάρχουν δύο συνθήκες τερματισμού της επαναληπτικής συνάρτησης, δηλαδή ο έλεγχος της ευκλείδειας νόρμας και ο μέγιστος αριθμός των επαναλήψεων. Αν η επαναληπτική συνάρτηση δεν τερματίσει με τον έλεγχο της νόρμας, επιστρέφεται σε όλες τις περιπτώσεις ο μέγιστος αριθμός επαναλήψεων. Στην αντίθετη περίπτωση χρησιμοποιείται η συνάρτηση colorAlgorithm().

Υπάρχουν αρκετοί αλγόριθμοι χρωματισμού και σε αυτό το σημείο θα περιγραφούν οι αλγόριθμοι που εμπεριέχονται στην συγκεκριμένη εφαρμογή.

5.1.1. Escape Time

Ο escape time algorithm ουσιαστικά επιστρέφει πόσες επαναλήψεις χρειάστηκαν, ώστε να τερματιστεί η επαναληπτική συνάρτηση με τον έλεγχο της ευκλείδειας νόρμας .

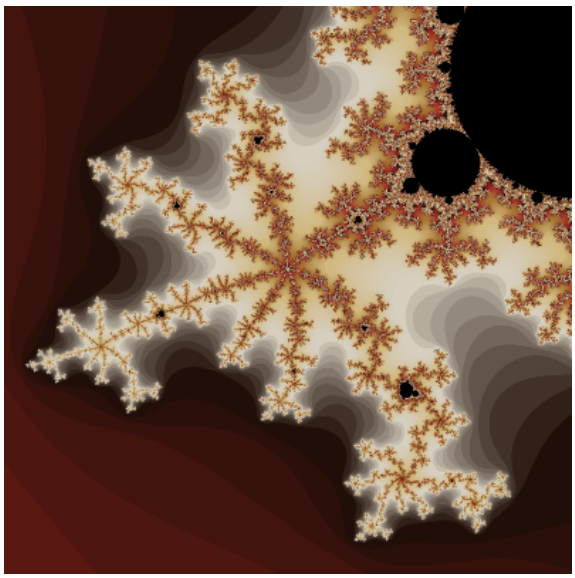
```
function double colorAlgorithm (iterations, ..,) {

    return iterations;

}
```

Στις Magnet functions υπάρχουν δύο διαφορετικοί έλεγχοι της ευκλείδειας νόρμας (άνω φράγματος και σύγκλισης), οπότε πρέπει να γνωρίζουμε, ποιά από τις δύο συνθήκες ίσχυε, για να επιστρέψουμε μια κατάλληλη τιμή.

```
function double colorAlgorithm (iterations,  bounded_condition, …,) {

    if(bounded_condition) {
        return iterations + 200;
    }
    else {
        return iterations;
    }

}
```

Η τιμή 200 στον παραπάνω ψευδοκώδικα είναι συμβολική, καθώς στη θέση της μπορεί να χρησιμοποιηθεί οποιαδήποτε τιμή. Ο μοναδικός σκοπός της είναι να έχουμε διαφορετικά χρώματα στις διαφορετικές περιπτώσεις τερματισμού της επαναληπτικής συνάρτησης.

Αποτέλεσμα του αλγορίθμου escape time:



5.1.2. Smooth

Σκοπός του συγκεκριμένου αλγορίθμου είναι να ελαττώσει τις απότομες εναλλαγές χρώματος που προκαλεί ο αλγόριθμος escape time, δημιουργώντας πιο ομαλές εικόνες. Οι τύποι και οι ψευδοκώδικες που ακολουθούν βρεθήκαν μετά από έρευνα στο διαδίκτυο. Αν και η ορθότητα τους δε δύναται να αποδειχθεί μαθηματικά, τουλάχιστον στα πλαίσια αυτής της εργάσιας, αυτό που απομένει είναι το αισθητικό αποτέλεσμα.

Για τις συναρτήσεις Mandelbrot, Mandelbrot Cubed, Mandelbrot Fourth, Mandelbrot Fifth, Mandelbrot Sixth, Mandelbrot Seventh, Mandelbrot Eighth, Mandelbrot Ninth, Mandelbrot Tenth, Mandelbrot Nth, Mandelbar, Manowar, Spider, Phoenix, την επιλογή Burning Ship, Lambda, Barnsley, Mandelbrot Poly και Sierpinski Gasket χρησιμοποιείται ο τύπος:

$$\text{iterations} + \frac{\ln(\text{bailout}) - \ln(|\text{Zn} - 1| + (1\text{e} - 8))}{\ln(|\text{Zn}|) - \ln(|\text{Zn} - 1| + (1\text{e} - 8))}$$

Όπου $|Z_n|$ είναι η ευκλείδεια νόρμα του μιγαδικού στο βήμα n, που τερμάτισε η επαναληπτική συνάρτηση και $|Z_{n-1}|$ η ευκλείδεια νόρμα στο βήμα n-1.

Φυσικά η τιμή ln(bailout) μπορεί να είναι προϋπολογισμένη, εξοικονομώντας υπολογιστικό φόρτο.

```
function double colorAlgorithm (iterations,  norm, bailout, norm_1
,…,) {

    return iterations + (ln(bailout) - ln(norm_1 + 1e-8)) / (ln(norm)
- ln(norm_1 + 1e-8));

}
```

Για τις συναρτήσεις Magnet χρησιμοποιείται ο τύπος:

Αν η επανάληψη τερματίσει με τη συνθήκη άνω φράγματος,

$$\text{iterations} + \frac{\ln(\text{bailout}) - \ln(|\text{c}| + (1\text{e} - 8))}{\ln(|\text{b}|) - \ln(|\text{c}| + (1\text{e} - 8))} + 200$$

Αν η επανάληψη τερματίσει με τη συνθήκη σύγκλισης,

$$\text{iterations} - \frac{\ln(\text{error}) - \text{a}}{\text{a} - \ln(|\text{b} - \text{c}|)}$$

Όπου a = $|Z_n - 1|$ από το κριτήριο σύγκλισης, b = Z_n και c = Z_{n-1}.

Όπου Z_n είναι ο μιγαδικός αριθμός στο βήμα n, που τερμάτισε η επαναληπτική συνάρτηση και Z_{n-1} ο μιγαδικός αριθμός στο βήμα n-1.

```
function double colorAlgorithm (iterations, bounded_condition, norm_1, norm, bailout, error, z, zold, …,) {

    if(bounded_condition) {
        return iterations + 200 + (ln(bailout) - ln(norm_1 + 1e-8)) / (ln(norm) - ln(norm_1 + 1e-8));
    }
    else {
        return iterations - (ln(error) - |z - 1| ) / (|z - 1| - ln(|z - zold|);
    }

}
```

Για την τιμή 200 στον παραπάνω ψευδοκώδικα, ισχύει το ίδιο, όπως και στον αλγόριθμο escape time για τις magnet functions.

Για τις Root Finding Methods χρησιμοποιείται ο τύπος:

$$\text{iterations} - \frac{\ln(\text{error}) - \ln(|b - c|)}{|a - b| - \ln(|b - c|)}$$

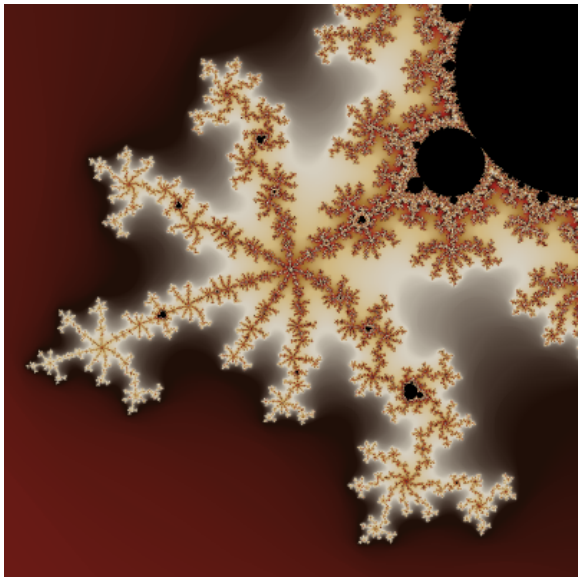
Όπου $a = Z_n$, $b = Z_{n-1}$ και $c = Z_{n-2}$.

```
function double colorAlgorithm (iterations, error, z, zold, zold2…,)
{

    return iterations - (ln(error) - ln(|zold - zold2|)) / (|z - zold| - ln(|zold - zold2|));

}
```

Όπου ln είναι ο φυσικός λογάριθμος (νεπέριος, συνήθως log στις math libraries).

Ο smooth algorithm είναι ο μόνος που ουσιαστικά χρειάζεται να επιστρέφει double τιμή. Όλοι οι άλλοι αλγόριθμοι τελικά κάνουν typecast σε int.

Αποτέλεσμα του αλγορίθμου smooth:



5.1.3. Binary Decomposition

Ο binary decomposition algorithm κάνει έναν έλεγχο στην τελευταία τιμή του μιγαδικού αριθμού στο βήμα n και συγκεκριμένα ελέγχει, αν το imaginary μέρος είναι μικρότερό του 0. Αν ισχύει αυτό, τότε προσθέτει μια σταθερή τιμή στις επαναλήψεις, αλλιώς επιστρέφει τον κανονικό αριθμό επαναλήψεων που χρειάστηκαν. Ουσιαστικά με αυτόν τον τρόπο πετυχαίνουμε μια διαφοροποίηση του χρώματος σε αυτές τις περιοχές.

```
function double colorAlgorithm (iterations,  z, …,) {

    if(Im(z) < 0 ) {
        return iterations + 50;
    }
    else {
        return iterations;
    }

}
```

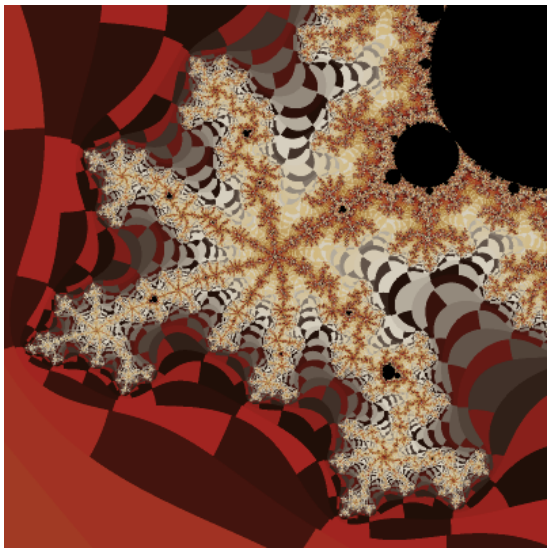
Για τις Magnet functions, όπως έχει ήδη αναφερθεί και για τον αλγόριθμο escape time πρέπει να ελέγχουμε το κριτήριο τερματισμού της επανάληψης (μιας και υπάρχουν δύο), ώστε διαφορετικό κριτήριο να αντιστοιχεί σε διαφορετικό χρώμα . Οπότε έχουμε,

```
function double colorAlgorithm (iterations,  bounded_condition, z,
…,) {

    if(bounded_condition) {
        if(Im(z) < 0 ) {
            return iterations + 50;
        }
```

```
        else {
            return iterations + 200;
        }
    }
    else {
        if(Im(z) < 0 ) {
            return iterations + 50;
        }
        else {
            return iterations;
        }
    }

}
```

Το 50 είναι απλά μια συμβολική τιμή.

Αποτέλεσμα του αλγορίθμου binary decomposition:



5.1.4. Binary Decomposition 2

Ο binary decomposition 2 algorithm κάνει έναν έλεγχο στην τελευταία τιμή του μιγαδικού αριθμού στο βήμα n και συγκεκριμένα ελέγχει, αν το real μέρος είναι μικρότερό του 0. Αν ισχύει αυτό τότε προσθέτει μια σταθερή τιμή στις επαναλήψεις, αλλιώς επιστρέφει τον κανονικό αριθμό επαναλήψεων που χρειάστηκαν. Ο συγκεκριμένος αλγόριθμος είναι μια παραλλαγή του binary decomposition.

```
function double colorAlgorithm (iterations,  z, …,) {

    if(Re(z) < 0 ) {
        return iterations + 50;
    }
    else {
        return iterations;
    }
```

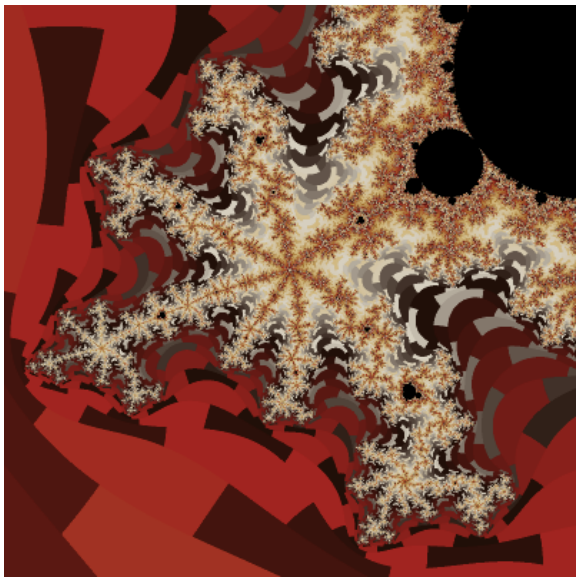
```
}
```

Για τις Magnet functions, όπως έχει ήδη αναφερθεί και για τον αλγόριθμο escape time πρέπει να ελέγχουμε το κριτήριο τερματισμού της επανάληψης (μιας και υπάρχουν δύο), ώστε διαφορετικό κριτήριο να αντιστοιχεί σε διαφορετικό χρώμα . Οπότε έχουμε,

```
function double colorAlgorithm (iterations,  bounded_condition, z, …,) {

    if(bounded_condition) {
        if(Re(z) < 0 ) {
            return iterations + 50;
        }
        else {
            return iterations + 200;
        }
    }
    else {
        if(Re(z) < 0 ) {
            return iterations + 50;
        }
        else {
            return iterations;
        }
    }

}
```

Το 50 είναι απλά μια συμβολική τιμή.

Αποτέλεσμα του αλγορίθμου binary decomposition 2:

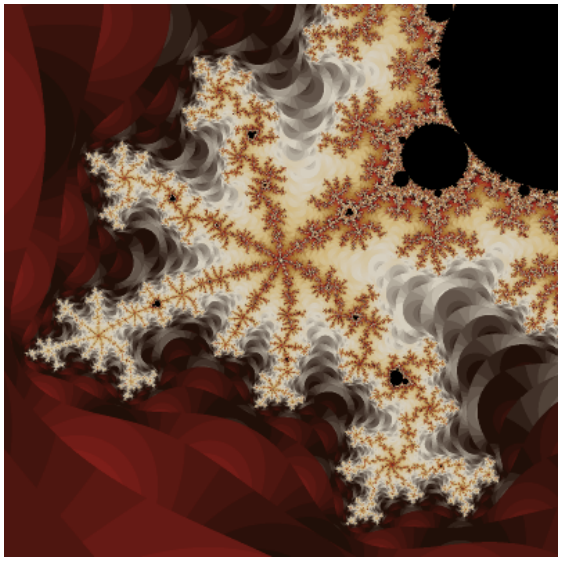


5.1.5. Escape Time + Re(z)

Ο escape time + Re(z), όπως άλλωστε δηλώνει και το όνομά του, προσθέτει στον αριθμό των επαναλήψεων το Re(z) της τελευταίας επανάληψης. Στη συνέχεια αυτός ο αριθμός γίνεται typecast σε int.

```
function double colorAlgorithm (iterations,  z, …,) {

    return (int)(iterations + Re(z));

}
```

Αποτέλεσμα του αλγορίθμου escape time + Re(z):

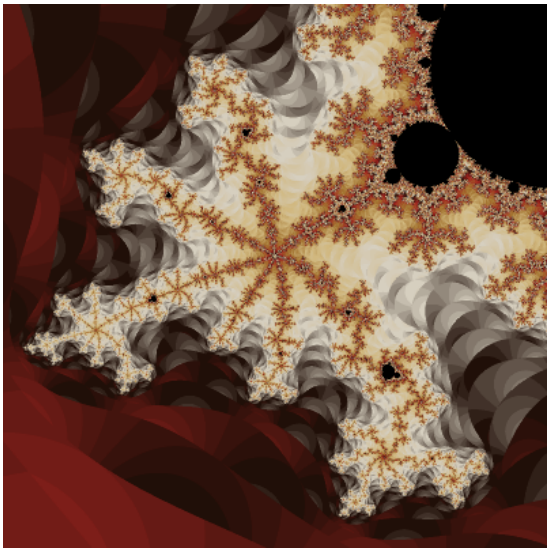


5.1.6. Escape Time + Im(z)

Ο escape time + Im(z), όπως άλλωστε δηλώνει και το όνομά του, προσθέτει στον αριθμό των επαναλήψεων το Im(z) της τελευταίας επανάληψης. Στη συνέχεια αυτός ο αριθμός γίνεται typecast σε int.

```
function double colorAlgorithm (iterations,  z, …,) {

    return (int)(iterations + Im(z));

}
```

Αποτέλεσμα του αλγορίθμου escape time + Im(z):

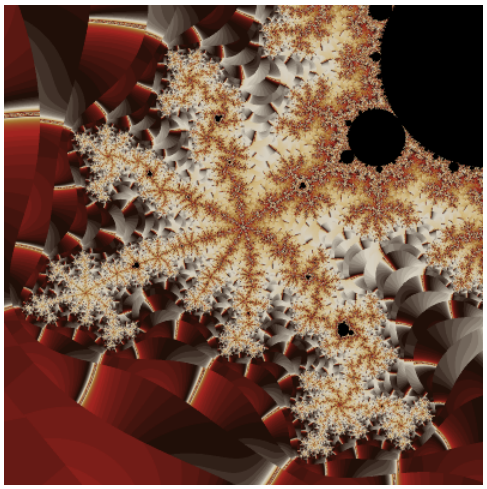


5.1.7. Escape Time + Re(z) + Im(z) + $\frac{Re(z)}{Im(z)}$

Ο escape time + Re(z) + Im(z) + $\frac{Re(z)}{Im(z)}$, όπως άλλωστε δηλώνει και το όνομά του, προσθέτει στον αριθμό των επαναλήψεων το Re(z), το Im(z) και το $\frac{Re(z)}{Im(z)}$ της τελευταίας επανάληψης. Στη συνέχεια αυτός ο αριθμός γίνεται typecast σε int.

```
function double colorAlgorithm (iterations,  z, …,) {

    return (int)(iterations + Re(z) + Im(z) + Re(z) / Im(z));

}
```

Αποτέλεσμα του αλγορίθμου escape time Re(z) + Im(z) + $\frac{Re(z)}{Im(z)}$:



5.1.8. Biomorph

Ο συγκεκριμένος αλγόριθμος πραγματοποιεί έναν πιο σύνθετο έλεγχο με βάση την τιμή του bailout, για να αποφασίσει για το τελικό χρωματικό αποτέλεσμα. Ο έλεγχος πραγματοποιείται στην τελευταία τιμή του z.

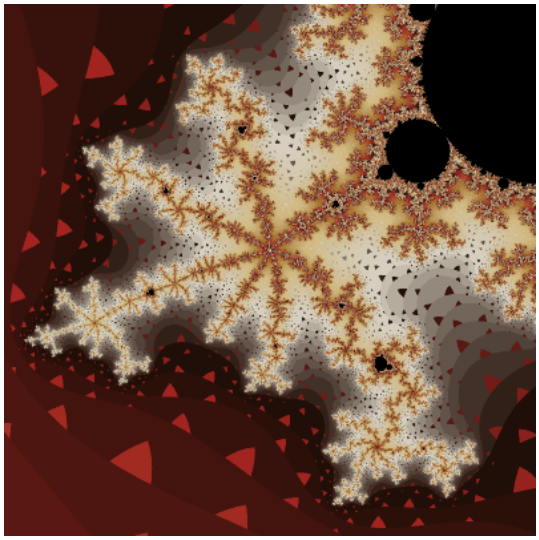
```
function double colorAlgorithm (iterations,  z, bailout, …,) {

    if((Re(z) > -bailout && Re(z) < bailout)||(Im(z) > -bailout &&
Im(z) < bailout)) {
        return iterations;
    }
    else {
        return iterations + 50;
    }

}
```

Για τις Magnet functions, όπως έχει ήδη αναφερθεί και για τον αλγόριθμο escape time πρέπει να ελέγχουμε το κριτήριο τερματισμού της επανάληψης (μιας και υπάρχουν δύο), ώστε διαφορετικό κριτήριο να αντιστοιχεί σε διαφορετικό χρώμα . Οπότε έχουμε,

```
function   double   colorAlgorithm   (iterations,      z,   bailout,
bounded_condition,…,) {

    if(bounded_condition) {
        if((Re(z) > -bailout && Re(z) < bailout)||(Im(z) > -bailout
&& Im(z) < bailout)) {
                return iterations + 200;
        }
        else {
                return iterations + 50;
        }
    }
    else {
        if((Re(z) > -bailout && Re(z) < bailout)||(Im(z) > -bailout
&& Im(z) < bailout)) {
                return iterations + 200;
        }
        else {
                return iterations;
        }
    }

}
```

Το 50 είναι απλά μια συμβολική τιμή.

Αποτέλεσμα του αλγορίθμου biomorph:



Επειδή ο αλγόριθμος biomorph κάνει ελέγχους στην τιμή του bailout (το οποίο χρησιμοποιείται μόνο στις επαναληπτικές συναρτήσεις που κάνουν έλεγχο άνω φράγματος), δε μπορούμε να το χρησιμοποιήσουμε στις Root Finding Functions καθώς κάνουν μόνο έλεγχο σύγκλισης.

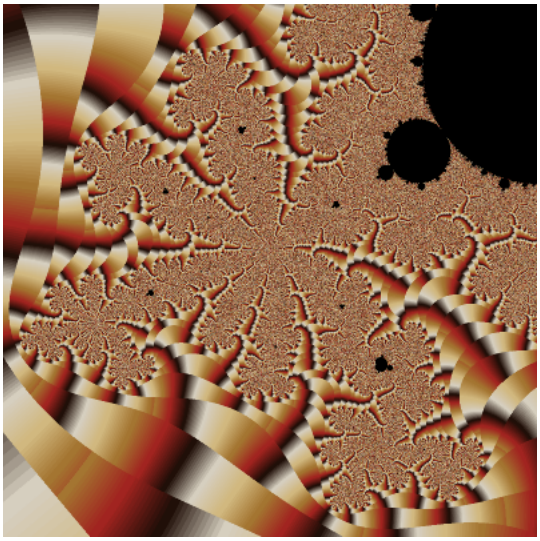
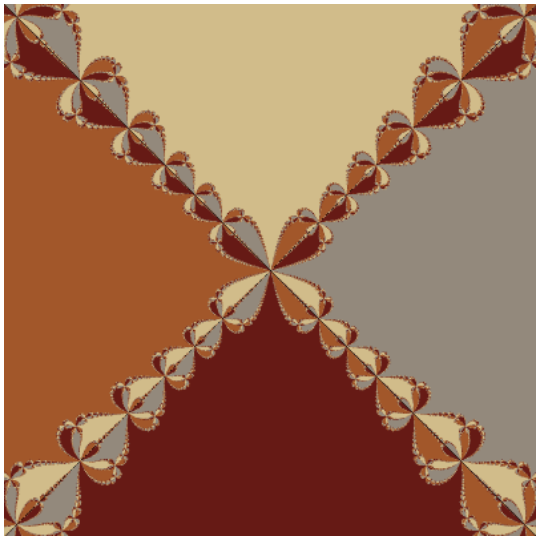
5.1.9. Color Decomposition

Ο color decomposition δεν χρησιμοποιεί καθόλου τον αριθμό των επαναλήψεων αλλά υπολογίζει μια τιμή βάσει του τύπου $(\frac{\text{atan2}(\text{Im}(z),\text{Re}(z))}{2\pi} + 0.75) * 59\pi$. Ο συγκεκριμένος αλγόριθμος έχει περισσότερο ενδιαφέρον στις Root Finding Methods, καθώς περιοχές που συγκλίνουν στη ίδια ρίζα έχουν το ίδιο χρώμα. Ο αλγόριθμος χρησιμοποιεί την τελευταία τιμή του μιγαδικού αριθμού στο βήμα n. Στο τέλος η τιμή γίνεται typecast σε int. Το Re(z) καθώς και Im(z) αντιστοιχούν στο μιγαδικό αριθμό της τελευταίας επανάληψης.

```
function double colorAlgorithm (z, …,) {

    return (int)((atan2(Im(z), Re(z)) / 2 * pi + 0.75) * 59 * pi);

}
```

Οι συντελεστές που πολλαπλασιάζονται με το atan2 απλά προσπαθούν να αυξήσουν την τιμή του συνολικού αποτελέσματος.

Αποτέλεσμα του αλγορίθμου color decomposition:

Όπως φαίνεται και από τις παραπάνω εικόνες, ο αλγόριθμος είναι πιο χρήσιμος στις Root Finding Functions.

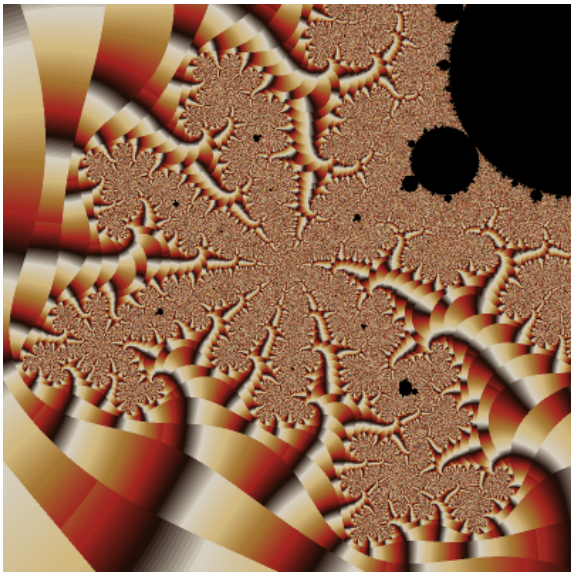
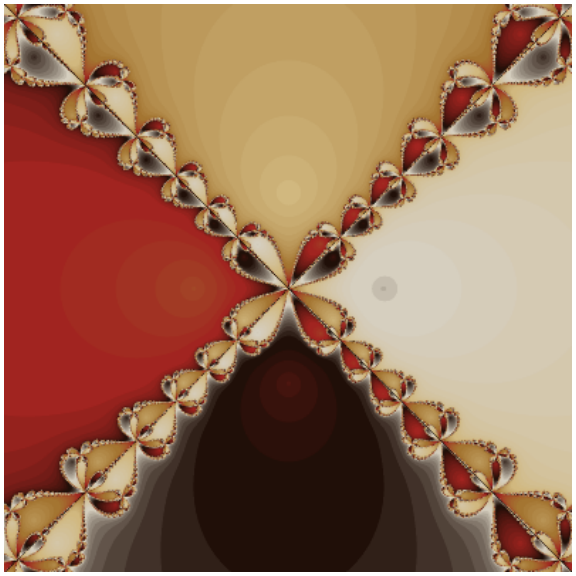
5.1.10. Escape Time + Color Decomposition

Ο escape time + color decomposition, όπως υποδηλώνει και το όνομά του, χρησιμοποίει τον αριθμό των επαναλήψεων καθώς και τον τύπο του color decomposition algorithm, δηλαδή $\text{iterations} + (\frac{\text{atan2}(\text{Im}(z),\text{Re}(z))}{2\pi} + 0.75) * 59\pi$. Ο συγκεκριμενος αλγοριθμος εχει περισσοτερο ενδιαφερον στις Root Finding Methods, καθόσον περιοχές που συγκλίνουν στη ίδια ρίζα θα έχουν το ίδιο χρώμα, αλλά επιπλέον θα λαμβάνεται υπ όψιν και ο αριθμός των επαναλήψεων, στον όποιο συνέβη η σύγκλιση στη ρίζα. Ο αλγόριθμος χρησιμοποιεί την τελευταία τιμή του μιγαδικού αριθμού στο βήμα n. Στο τέλος η τιμή γίνεται typecast σε int. Το Re(z) καθώς και Im(z) αντιστοιχούν στο μιγαδικό αριθμό της τελευταίας επανάληψης.

```
function double colorAlgorithm (iterations, z, …,) {

    return (int)(iterations + (atan2(Im(z), Re(z)) / 2 * pi + 0.75) *
59 * pi);

}
```

Οι συντελεστές που πολλαπλασιάζονται με το atan2 απλά προσπαθούν να αυξήσουν την τιμή του συνολικού αποτελέσματος.

Αποτέλεσμα του αλγορίθμου escape time + color decomposition:

Όπως φαίνεται και από τις παραπάνω εικόνες, ο αλγόριθμος είναι πιο χρήσιμος στις Root Finding Functions.

5.2. Δημιουργία παλέτας

Οι παλέτες από προγραμματιστική σκοπιά είναι απλά πίνακες που περιέχουν τις τιμές των χρωμάτων. Το γέμισμα όμως των πινάκων με τυχαίες τιμές χρώματος δεν θα είχε καλό αισθητικό αποτέλεσμα, για το λόγο αυτό, επιλέγουμε έναν αριθμό από βασικά χρώματα που θα περιέχονται στην παλέτα, ενώ όλα τα υπόλοιπα χρώματα υπολογίζονται με τη μέθοδο της γραμμικής παρεμβολής (linear interpolation).

Έστω ότι έχουμε δύο χρώματα, το χρώμα A και το χρώμα B. Με τη χρήση της γραμμικής παρεμβολής μπορούμε να μεταβούμε από το ένα χρώμα στο άλλο με αρκετά ομαλό τρόπο. Δεδομένου ότι, τα χρώματα A και B αποτελούνται από τιμές red, green, blue και επειδή θέλουμε να δημιουργήσουμε ένα αριθμό από n αποχρώσεις μεταξύ τους, οι τύποι της γραμμικής παρεμβολής είναι οι ακόλουθοι:

$$C.red = A.red + (B.red - A.red) * i$$

$$C.green = A.green + (B.green - A.green) * i$$

$$C.blue = A.blue + (B.blue - A.blue) * i$$

Όπου C είναι το αποτέλεσμα χρώμα και το i παίρνει τιμές μεταξύ 0 και 1, ενώ κάθε υποδιαίρεση του i έχει μέγεθος $\frac{1}{n}$.

Παραδείγματος χάριν, εάν ο χρήστης κάνει την παρακάτω επιλογή στον Custom Palette Editor,

Το αποτέλεσμα θα είναι 7 βασικά χρώματα ∗ 3 αποχρώσεις το καθένα, σύνολο 21 χρώματα, τα ακόλουθα:

Η εφαρμογή περιέχει έναν αριθμό από προϋπολογισμένες παλέτες. Φυσικά ο κάθε χρήστης μπορεί να δημιουργήσει τις δικές του παλέτες και να τις αποθηκεύσει / ξαναφορτώσει.

5.3. Χρήση παλέτας

Έχοντας πλέον καθορίσει τις επαναληπτικές συναρτήσεις, τους αλγόριθμους χρωματισμού καθώς και τη δομή της παλέτας, είμαστε πλέον σε θέση να υλοποιήσουμε τη συνάρτηση getColor(), την οποία χρησιμοποιήσαμε κατά τη δημιουργία της εικόνας. Ο ψευδοκώδικας που δημιουργούσε την εικόνα ήταν ο ακόλουθος:

```
function drawImage()  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            complex = convertPixeltoComplex(i,  j);
            value = iterativeFunction(complex);
            color = getColor(value);
            paintPixel(image, i, j, color);
        }
    }

}
```

Η getColor() θα πρέπει να υλοποιηθεί με τον ακόλουθο τρόπο:

```
function Color getColor (value) {

    if(value == max_iterations) {
        return fractal_color;
    }
    else {
        return palette[value % palette_size];
```

```
    }

}
```

Η μεταβλητή fractal_color (default τιμή μαύρο χρώμα) αντιστοιχεί στο μέγιστο αριθμό επαναλήψεων και μπορεί να επιλεχθεί από το χρήστη σε κάποια άλλη τιμή χρώματος. Ο πίνακας palette περιέχει τις τιμές χρώματος που προέκυψαν μετά από τη γραμμική παρεμβολή. Με τη χρήση του % (modulo) μπορούμε χρησιμοποιούμε κυκλικά την παλέτα.

5.4. Συχνότητα των χρωμάτων

Μπορούμε να αλλάξουμε τη συχνότητα εμφάνισης των χρωμάτων της παλέτας, είτε σε πιο μικρή είτε σε πιο μεγάλη, πολλαπλασιάζοντας το value με κάποιο συντελεστή.

Οπότε ο ψευδοκώδικας της getColor() μεταβάλλεται σε:

```
function Color getColor (value) {

    if(value == max_iterations) {
        return fractal_color;
    }
    else {
        return palette[(value * coefficient) % palette_size];
    }

}
```

5.5. Αλλαγή παλέτας

Ένας τρόπος για να αλλάξουμε την επιλεγμένη παλέτα σε κάποια διαφορετική, είναι να κάνουμε πάλι τα παραπάνω βήματα, δηλαδή να υπολογίσουμε όλη την εικόνα με την επαναληπτική συνάρτηση και στη συνέχεια, κατά τον καθορισμό των χρωμάτων στη getColor(), η εικόνα χρωματίζεται με διαφορετικό τρόπο, μιας και ο πίνακας palette περιέχει άλλα χρώματα.

Το πρόβλημα με την παραπάνω μέθοδο είναι ότι ο υπολογισμός της εικόνας μέσω της επαναληπτικής συνάρτησης είναι χρονοβόρος, οπότε στην περίπτωσή μας, εξυπηρετεί να κάνουμε caching των αποτελεσμάτων.

Οπότε ο ψευδοκώδικας δημιουργίας της εικόνας μεταβάλλεται σε:

Δηλώνουμε μέσα στην εφαρμογή μας την cache,

```
image_values[width][height];
```

```
function drawImage()  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            complex = convertPixeltoComplex(i,  j);
            image_values[i][j] = iterativeFunction(complex);
            color = getColor(image_values[i][j]);
            paintPixel(image, i, j, color);
        }
    }

}
```

Έτσι ο ψευδοκώδικας για την αλλαγή παλέτας είναι ο ακόλουθος:

```
function changePalette()  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            color = getColor(image_values[i][j]);
            paintPixel(image, i, j, color);
        }
    }

}
```

5.6. Μετακίνηση μέσα στην παλέτα (Palette Shifting / Color Cycling)

Εάν θέλουμε να μετακινήσουμε τα χρώματα της ενεργής παλέτας (μπροστά ή πίσω) χρειάζεται να έχουμε μια μεταβλητή offset, στην οποία θα αλλάζουμε τις τιμές. Η μεταβλητή offset σε συνδυασμό με τις cached τιμές των επαναληπτικών συναρτήσεων, καθώς και με το γεγονός ότι η παλέτα μας είναι κυκλική (χρήση του modulo), μας επιτρέπει να πετύχουμε αυτό το αποτέλεσμα, δηλαδή της μετακίνησης και της εναλλαγής των χρωμάτων.

Οπότε οι ψευδοκώδικες για τη δημιουργία της εικόνας, για την αλλαγή παλέτας και η getColor() αλλάζουν ως εξής:

```
function drawImage()  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            complex = convertPixeltoComplex(i,  j);
            image_values[i][j] = iterativeFunction(complex);
```

```
            color = getColor(image_values[i][j], offset);
            paintPixel(image, i, j, color);
        }
    }

}
```

```
function changePalette()  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            color = getColor(image_values[i][j], offset);
            paintPixel(image, i, j, color);
        }
    }

}
```

```
function Color getColor (value, offset) {

    if(value == max_iterations) {
        return fractal_color;
    }
    else {
        return   palette[((value   +   offset)   *   coefficient)   %
palette_size];
    }

}
```

Οι τιμές του offset μεταβάλλονται από το χρήστη.

Για το color cycling χρησιμοποιείται ίδια λογική, όπως και παραπάνω, μόνο που σε αυτή την περίπτωση το offset αλλάζει αυτόματα, ενώ η διαδικασία είναι επαναληπτική για να προσδώσει την αίσθηση του animation.

Ο ψευδοκώδικας που το υλοποιεί είναι ο ακόλουθος:

```
function  colorCycling() {

    do {
        offset++;

        for(i = 0; i < height; i++) {
            for(j = 0; j < width; j++) {
                color = getColor(image_values[i][j], offset);
                paintPixel(image, i, j, color);
            }
        }

    } while (!user intervention);
}
```

6. Μεγέθυνση – Σμίκρυνση (Zoom in – Zoom out)

Μέχρι τώρα έχουμε ολοκληρώσει τα βασικά στάδια για τη δημιουργία μιας εικόνας. Αυτό όμως που κάνει τις εφαρμογές απεικόνισης fractals να ξεχωρίζουν, είναι η δυνατότητα που δίνουν στο χρήστη να μεγεθύνει, όποιες περιοχές θέλει. Με αυτό τον τρόπο ο χρήστης μπορεί να «ταξιδεύει» μέσα στην εικόνα!

Φυσικά δεν είναι μόνο αυτή η δυνατότητα που κάνει τις εφαρμογές να ξεχωρίζουν. Θα πρέπει να μπορούν να κάνουν zoom στην εικόνα σε real time χρόνο, καθώς και να φτάνουν σε πολύ μεγάλα επίπεδα μεγέθυνσης.

Το γεγονός όμως ότι η εφαρμογή γράφτηκε σε Java, αυτομάτως αποκλείει και τα δύο παραπάνω ενδεχόμενα, μιας και η Java έχει μεγάλα προβλήματα απόδοσης και επίσης για να φτάσουμε σε μεγαλύτερα επίπεδα μεγέθυνσης θα έπρεπε να χρησιμοποιούμε αριθμούς μεγαλύτερης ακρίβειας από τους double. Η Java δεν έχει κάποιο primitive τύπο καλύτερης ακρίβειας σε σχέση με τους double, οπότε θα πρέπει να κατασκευάσουμε κάποιο τύπο που συνδυάζει περισσότερους του ενός double αριθμούς για να επιτύχει ακρίβεια. Αυτό όμως σημαίνει ότι απαιτούνται και περισσότερες πράξεις για κάθε αριθμό, γεγονός που εντείνει τα προβλήματα απόδοσης της Java.

Συνεπώς, αφού τα δυο προβλήματα σχετίζονται και η λύση τους στη Java είναι σχεδόν αδύνατη, για την υλοποίηση χρησιμοποιούνται double αριθμοί.

Ξαναγυρίζουμε στη συνάρτηση convertPixelToComplex(),

```
function Complex convertPixeltoComplex(int i,  int j) {

    real = real_center - size / 2 + j * (size / width);
    imaginary = imaginary_center - size / 2 + i * (size / height);
    return Complex(real, imaginary);

}
```

Η τιμή που σχετίζεται με τη μεγέθυνση και τη σμίκρυνση μιας περιοχής είναι το size. Όταν αυξάνεται το size, τότε έχουμε σμίκρυνση στην εικόνα, ενώ όταν μειώνεται το size, έχουμε μεγέθυνση στην εικόνα. Ο χρήστης μπορεί να προσδιορίσει το ρυθμό μεγέθυνσης/σμίκρυνσης (zooming factor, default τιμή 2), ο οποίος στη συνέχεια πολλαπλασιάζεται/διαιρείται με το size για να προκύψει το νέο size.

Για μεγέθυνση της εικόνας έχουμε, size = size / zooming_factor, ενώ για σμίκρυνση της εικόνας έχουμε, size = size * zooming_factor.

Στην εφαρμογή, zoom, γίνεται με δύο τρόπους. Είτε με αριστερό/δεξί mouse click (zoom in/zoom out) επιλέγοντας σαν νέο κέντρο το pixel που επιλέχτηκε, είτε με τα κουμπιά του πληκτρολογίου +/- (zoom in/zoom out) κρατώντας σαν κέντρο το παλιό. Στο

τέλος καλούμε τη συνάρτηση drawImage() για να υπολογιστεί η νέα μας εικόνα. Ο ψευδοκωδικας για τις δυο υλοποιήσεις είναι ο εξής:

```
function  mouseClicks(int x,  int y, button) {
    real_center = real_center - size / 2 + y * (size / width);
    imaginary_center = imaginary_center - size / 2 + x * (size /
height);

    if(button == LEFT) {
        size = size / zooming_factor;
    }
    else if(button == RIGHT) {
        size = size * zooming_factor;
    }

    drawImage();
}
```

Στον παραπάνω ψευδοκώδικα, παρατηρούμε ότι οι ακέραιες τιμές των συντεταγμένων του pixel μετατρέπονται σε πραγματικούς αριθμούς, ακριβώς με τον ίδιο τρόπο που περιγράφθηκε στη συνάρτηση convertPixelToComplex().

```
function  keyboard(key) {

    if(key == '+') {
        size = size / zooming_factor;
    }
    else if(key == '-') {
        size = size * zooming_factor;
    }

    drawImage();
}
```

7. Μετασχηματισμοί του αρχικού pixel

Μπορούμε να εφαρμόσουμε διάφορους μετασχηματισμούς στην αρχική τιμή, που προέκυψε από την αντιστοίχηση του pixel σε μιγαδικό αριθμό. Σκοπός του μετασχηματισμού είναι να δημιουργηθεί κάποια διαφορετική απεικόνιση, χρησιμοποιώντας τον ίδιο επαναληπτικό κανόνα.

Θα δείξουμε συγκεκριμένα την αλλαγή του ψευδοκώδικα για τη συνάρτηση Mandelbrot, διότι ο ψευδοκώδικας για όλες τις άλλες συναρτήσεις αλλάζει με όμοιο τρόπο.

```
function double fractalFunction(Complex pixel) {
    pixel = applyTransformation(pixel);
    z = pixel;
    c = pixel;
```

```
    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
    }

    return max_iterations;
}
```

Η συνάρτηση juliaFunction() δε χρειάζεται κάποια αλλαγή μέσα στον ψευδοκώδικά της. Η μόνη αλλαγή που πρέπει να γίνει, είναι στην επιλογή του seed, δηλαδή, όταν ο χρήστης επιλέξει κάποιο seed και έχει ενεργοποιημένο κάποιο μετασχηματισμό, η τιμή του seed αλλάζει σε, seed = applyTransformation(seed). Ορισμένοι γνωστοί μετασχηματισμοί είναι οι εξής:

7.1. μ Plane

Η αρχική τιμή παραμένει ως έχει (default επιλογή).

```
function Complex applyTransformation (Complex z) {

    return z;

}
```

7.2. $\frac{1}{\mu}$ Plane

Η αρχική τιμή υπόκειται τον μετασχηματισμό $z = \frac{1}{z}$.

```
function Complex applyTransformation (Complex z) {

    return 1 / z;

}
```

7.3. $\frac{1}{\mu+0.25}$ Plane

Η αρχική τιμή υπόκειται τον μετασχηματισμό $z = \frac{1}{z+0.25}$.

```
function Complex applyTransformation (Complex z) {

    return 1 / (z + 0.25);
```

```
}
```

7.4. $\frac{1}{\mu-1.40115}$ Plane

Η αρχική τιμή υπόκειται τον μετασχηματισμό $z = \frac{1}{z-1.40115}$.

```
function Complex applyTransformation (Complex z) {
    return 1 / (z - 1.40115);
}
```

7.5. $\frac{1}{\mu-2}$ Plane

Η αρχική τιμή υπόκειται τον μετασχηματισμό $z = \frac{1}{z-2}$.

```
function Complex applyTransformation (Complex z) {
    return 1 / (z - 2);
}
```

7.6. λ Plane

Η αρχική τιμή υπόκειται τον μετασχηματισμό $z = z * (1 - z)$.

```
function Complex applyTransformation (Complex z) {
    return z * (1 - z);
}
```

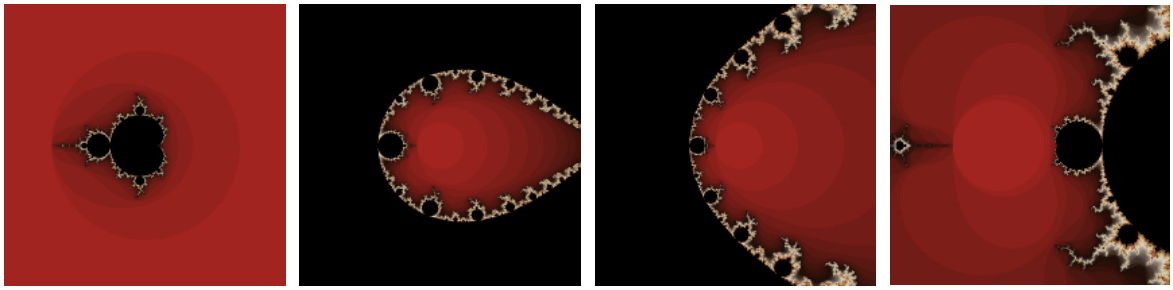
7.7. $\frac{1}{\lambda}$ Plane

Η αρχική τιμή υπόκειται τον μετασχηματισμό $z = \frac{1}{z * (1 - z)}$.

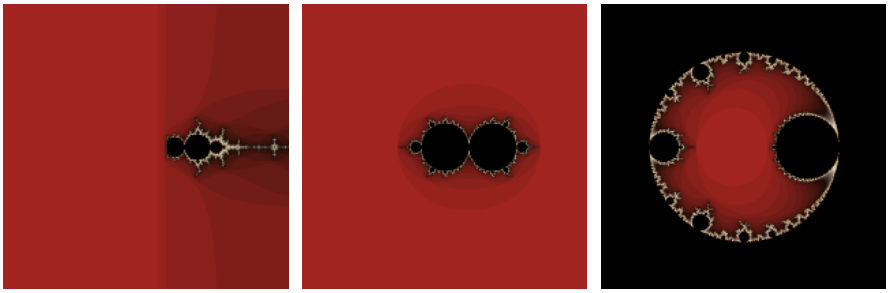
```
function Complex applyTransformation (Complex z) {
```

```
        return 1 / (z * (1 - z));

}
```

Παραδείγματα μετασχηματισμού αρχικού pixel του Mandelbrot set:



μ $\frac{1}{\mu}$ $\frac{1}{\mu+0.25}$ $\frac{1}{\mu-1.40115}$



$\frac{1}{\mu-2}$ λ $\frac{1}{\lambda}$

8. Περιστροφή (Rotation)

Δίνεται η δυνατότητα στο χρήστη να περιστρέψει την εικόνα, γύρω από το σημείο (0, 0), απλά επιλέγοντας τη γωνία περιστροφής σε μοίρες, δηλαδή από -360° έως 360°.

Η περιστροφή στην πράξη είναι και αυτή ένας μετασχηματισμός. Δεδομένου ενός σημείου στο δισδιάστατο επίπεδο (x, y) η περιστροφή του θ μοίρες, γύρω από το σημείο του επίπεδου (0, 0), μεταφέρει το αρχικό σημείο στη θέση (x', y'). Η περιστροφή πραγματοποιείται από την ακόλουθη πράξη:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \longrightarrow \begin{aligned} x' &= x\cos\theta - y\sin\theta \\ y' &= x\sin\theta + y\cos\theta. \end{aligned}$$

Στους μιγαδικούς αριθμούς μπορούμε να έχουμε το ίδιο αποτέλεσμα, πολλαπλασιάζοντας το μιγαδικό αριθμό x + yi με το μιγαδικό αριθμό cosθ + sinθi.

$$
\begin{aligned}
e^{i\theta}z &= (\cos\theta + i\sin\theta)(x + iy) \\
&= (x\cos\theta + iy\cos\theta + ix\sin\theta - y\sin\theta) \\
&= (x\cos\theta - y\sin\theta) + i(x\sin\theta + y\cos\theta) \\
&= x' + iy',
\end{aligned}
$$

Θα δείξουμε συγκεκριμένα την αλλαγή του ψευδοκώδικα για τη συνάρτηση Mandelbrot, διότι ο ψευδοκώδικας για όλες τις άλλες συναρτήσεις αλλάζει με όμοιο τρόπο.

```
function double fractalFunction(Complex pixel) {
    pixel = applyRotation(pixel);
    pixel = applyTransformation(pixel);
    z = pixel;
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
    }

    return max_iterations;
}
```

Η συνάρτηση juliaFunction(), σε αντίθεση με τους μετασχηματισμούς του αρχικού pixel, θα χρειαστεί αλλαγή στον ψευδοκώδικά της, καθώς επίσης θα χρειαστεί να γίνει αλλαγή και στην επιλογή του seed. Δηλαδή, όταν ο χρήστης επιλέξει κάποιο seed και έχει ενεργοποιημένη την επιλογή της περιστροφής, η τιμή του seed αλλάζει σε, seed = applyRotation(seed).

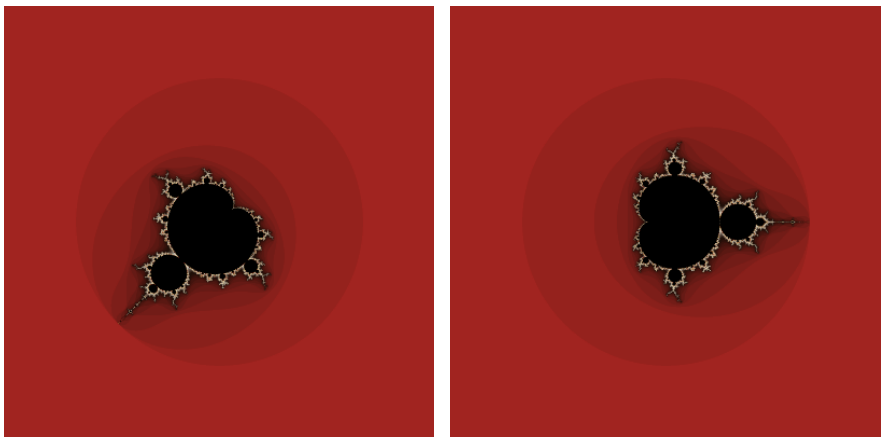
```
function double juliaFunction(Complex pixel) {
    pixel = applyRotation(pixel);
    z = pixel;
    c = seed;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
    }

    return max_iterations;
```

Ακολουθεί ο ψευδοκώδικας για τη συνάρτηση applyRotation():

```
function Complex applyRotation (Complex z) {

    return z * rotation;

}
```

Όπου η μεταβλητή rotation = Complex(cos(theta), sin(theta)) μπορεί να έχει προϋπολογιστεί, ώστε να μην επιβαρύνεται η επαναληπτική συνάρτηση με τον υπολογισμό των τριγωνομετρικών συναρτήσεων.

Παραδείγματα περιστροφής του Mandelbrot set:



45°

180°

9. Ορισμός αρχικής τιμής (Perturbation)

Σε όλες τις επαναληπτικές συναρτήσεις η αρχική τιμή του z είναι η τιμή του pixel, ή κάποιος μετασχηματισμός (μετασχηματισμοί αρχικού pixel ή περιστροφή) της τιμής του pixel (εκτός των Lambda και Magnet, οι οποίες αρχικοποιούνται στους μιγαδικούς αριθμούς 0.5 + 0.0i και 0.0 + 0.0i αντίστοιχα) . Αυτή η επιλογή δίνει την δυνατότητα στο χρήστη να προσδιορίσει αυτή την αρχική τιμή, θέτοντας ένα μιγαδικό αριθμό.

Η ανάθεση αυτή της αρχικής τιμής δεν μπορεί να συμβεί στις Root Finding Methods, στη συνάρτηση Sierpinski Gasket, καθώς και στα Julia sets οποιασδήποτε συνάρτησης, διότι όλα τα προαναφερθέντα έχουν μόνο ένα βαθμό ελευθερίας, οπότε δε μπορούμε να χρησιμοποιήσουμε μια σταθερή τιμή για να δημιουργήσουμε μια εικόνα.

Θα δείξουμε συγκεκριμένα την αλλαγή του ψευδοκώδικα για τη συνάρτηση Mandelbrot, καθόσον ο ψευδοκώδικας για όλες τις άλλες συναρτήσεις αλλάζει με όμοιο τρόπο.

```
function double fractalFunction(Complex pixel) {
    pixel = applyRotation(pixel);
    pixel = applyTransformation(pixel);
    z = applyPerturbation(pixel);
    c = pixel;

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(|z| >= bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
    }

    return max_iterations;
}
```

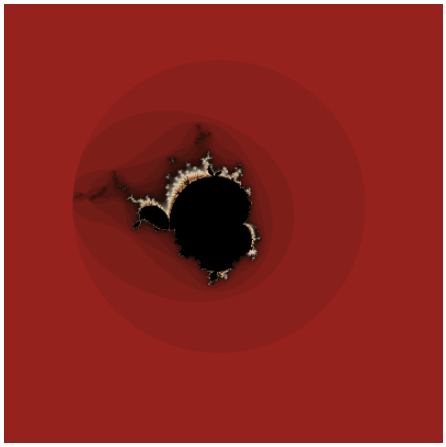
Ακολουθεί ο ψευδοκώδικας για τη συνάρτηση applyPerturbation():

```
function Complex applyPerturbation (Complex z) {

    if(perturbation_is_enabled) {
        return perturbation;
    }
    else {
        return z;
    }

}
```

Όπου η μεταβλητή perturbation_is_enabled ενεργοποιείται από το χρήστη, ενώ η μεταβλητή perturbation περιέχει το μιγαδικό αριθμό που έδωσε ο χρήστης. Ο παραπάνω ψευδοκώδικας στην περίπτωση της Lambda ή των Magnet functions αντί του return z έχουν return Complex(0.5, 0) , return Complex(0, 0) αντίστοιχα.

Παραδείγματα ορισμού αρχικής τιμής του Mandelbrot set:

0.5



$0.4 - 0.3i$

10. Τροχιά ενός μιγαδικού αριθμού (Orbit)

Ένα πολύ σημαντικό εργαλείο της εφαρμογής είναι η δυνατότητα οπτικοποίησης της τροχιάς ενός μιγαδικού αριθμού κατά την εκτέλεση οποιασδήποτε επαναληπτικής συνάρτησης πάνω σε αυτόν.

Η συγκεκριμένη λειτουργία δουλεύει με τον ακόλουθο τρόπο. Αρχικά ο χρήστης επιλέγει, με αριστερό mouse click, ένα pixel της εικόνας. Στη συνέχεια αυτό μετατρέπεται βάσει της συνάρτησης convertPixeltoComplex() σε ένα μιγαδικό αριθμό και προστίθεται σε μια λίστα με μιγαδικούς αριθμούς. Κατόπιν αυτός ο αριθμός περνά από όλα τα στάδια μιας επαναληπτικής συνάρτησης, είτε η συνάρτηση είναι fractalFunction(), είτε είναι juliaFunction(), χρησιμοποιώντας, όποιους μετασχηματισμούς είναι ενεργείς. Η μοναδική διαφορά είναι ότι η επαναληπτική συνάρτηση δεν κάνει κανέναν έλεγχο νόρμας (άνω φράγματος ή σύγκλισης) αλλά εκτελείται για τον μέγιστο αριθμό επαναλήψεων, μιας και για τη συγκεκριμένη λειτουργία αυτός ο έλεγχος δεν έχει κάποια σημασία. Σε κάθε βήμα η νέα τιμή του μιγαδικού προστίθεται στη λίστα με τους μιγαδικούς. Στο τέλος διατρέχουμε όλους τους μιγαδικούς αριθμούς της λίστας και χρησιμοποιούμε τη συνάρτηση convertComplexToPixel() (αντίστροφη της convertPixeltoComplex()). Χρησιμοποιούμε ανά δυο τις τιμές των pixel και ζωγραφίζουμε μια γραμμή μεταξύ τους. Με αυτόν τον τρόπο έχουμε οπτικοποιήσει την τροχιά που ακολούθησε ο συγκεκριμένος μιγαδικός αριθμός.

Θα δείξουμε συγκεκριμένα τον ψευδοκώδικα για τη συνάρτηση Mandelbrot, χωρίς να χρησιμοποιήσουμε κάποιο μετασχηματισμό (καθόσον η λογική είναι ανάλογη με τις απλές περιπτώσεις), καθώς επίσης και τους ψευδοκώδικες για το ζωγράφισμα της γραμμής και της συνάρτησης convertComplexToPixel().

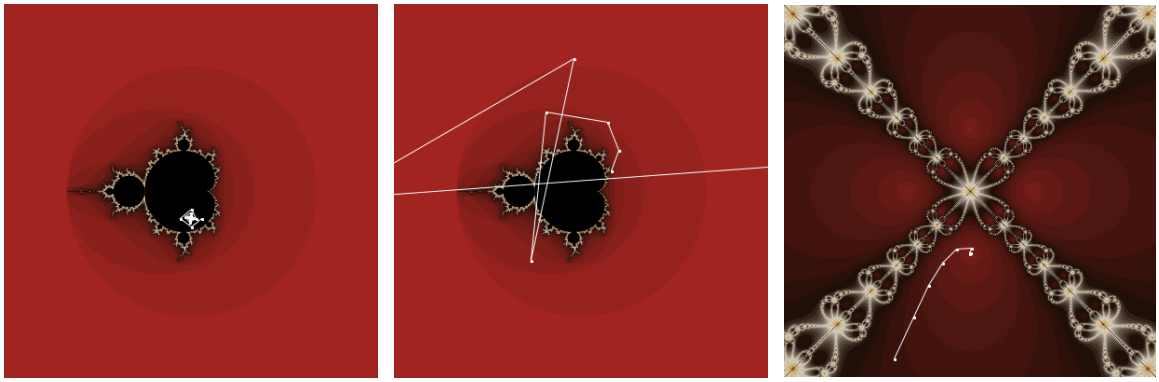
```
function Complex List fractalOrbit(Complex pixel) {
    z = pixel;
    c = pixel;
    list.add(z);

    for(iterations = 0; iterations < max_iterations; iterations++) {
        z = z^2 + c;
        list.add(z);
    }

    return list;
}
```

```
function Complex List juliaOrbit(Complex pixel) {
    z = pixel;
    c = seed;
    list.add(z);

    for(iterations = 0; iterations < max_iterations; iterations++) {
        z = z^2 + c;
        list.add(z);
    }

    return list;
}
```

```
function drawOrbitLine(Complex List list) {

    for(i = 0; i < list.size - 1; i++) {
        pixel1 = convertComplextoPixel(list.get(i));
        pixel2 = convertComplextoPixel(list.get(i + 1));
        drawLine(pixel1, pixel2);
    }

}
```

```
function int[] convertComplextoPixel(Complex z) {

    j =  (z.real - real_center + size / 2) / (size / width);
    i =   (z.imaginary  -  imaginary_center  +  size  /  2)  /  (size  /
height);
    return [i, j];

}
```

Παραδείγματα τροχιάς:



11. Βελτιστοποιήσεις

Όπως έχει ήδη αναφερθεί η Java έχει σημαντικά προβλήματα απόδοσης στο χρόνο εκτέλεσης, και γενικά δε χρησιμοποιείται σε εφαρμογές που προσπαθούν να ελαχιστοποιήσουν το χρόνο εκτέλεσής τους. Οι λόγοι που η συγγραφή του κώδικα συνεχίστηκε στη Java ήταν ότι:

1. Η αρχική εργασία που ζητήθηκε ήταν σε Java.
2. Ήταν αρκετά εύκολο να δημιουργηθεί το Graphic User Interface.
3. Δεν είχε υπολογιστεί ότι η εφαρμογή θα έφτανε σε τέτοιο επίπεδο, οπότε ήταν προτιμότερη η απλότητα.

Σχετικά γρήγορα εντοπίσθηκε, ότι ο χρόνος για τον υπολογισμό μερικών εικόνων ήταν αρκετά μεγάλος (ιδιαίτερα αυτών που περιέχουν πολλές μαύρες περιοχές / μέγιστος αριθμός επαναλήψεων), οπότε δοκιμάσθηκε ένας μεγάλος αριθμός βελτιστοποιήσεων. Για να υπάρχει κάποιος βαθμός σύγκρισης, παρατίθεται ο χρόνος υπολογισμού της αρχικής εικόνας του Mandelbrot set (size = 6, center = 0 + 0i, max_iterations = 3000, N = 2000) σύμφωνα με τους ψευδοκώδικες που έχουν μέχρι τώρα περιγραφεί. Οι προδιαγραφές του υπολογιστή, στον οποίο έγιναν οι μετρήσεις, δεν έχουν νόημα σε αυτό το σημείο, καθόσον οι χρόνοι δίνονται για τη μεταξύ τους σύγκριση.

Χρόνος υπολογισμού της εικόνας: 9095 ms.

Η συγκεκριμένη έκδοση της εφαρμογής συμπεριλαμβάνει τις εξής βελτιστοποιήσεις:

- Norm Squared
- Threads
- Periodicity Checking
- Boundary Tracing

Στη συνέχεια ακολουθεί η ανάλυση ορισμένων τύπων βελτιστοποιήσεων.

11.1. Βελτιστοποίηση υπολογισμών

11.1.1. Norm Squared

Ένας από τους σημαντικότερους λόγους καθυστέρησης είναι η χρήση συναρτήσεων από τη math library, όπως για παράδειγμα η χρήση τριγωνομετρικών συναρτήσεων. Μια ακόμη κρυμμένη χρήση συνάρτησης από τη math library, είναι η χρήση της τετραγωνικής ρίζας (sqrt). Για τις συνθήκες τερματισμού των επαναληπτικών συναρτήσεων χρειαζόμαστε τον υπολογισμό της ευκλείδειας νόρμας του μιγαδικού αριθμού, δηλαδή $|Z| \geq \text{bailout}$ ή $|Z - Zold| \leq \text{error}$. Αναλύουμε λοιπόν την πρώτη περίπτωση, αφού και οι δυο έλεγχοι κάνουν ουσιαστικά το ίδιο πράγμα.

Αν $Z = a + bi$ τότε $|Z| = \sqrt{a^2 + b^2}$, αρα εχουμε $\sqrt{a^2 + b^2} \geq bailout$. Το προηγούμενο υψώνοντας και τα δύο μέλη στο τετράγωνο, είναι ισοδύναμο με $a^2 + b^2 \geq bailout^2$, $a * a + b * b \geq bailout * bailout$. Με αυτό τον τρόπο, σαν συνθήκη τερματισμού, ελέγχουμε κάτι ισοδύναμο, χωρίς να χρειάζεται να χρησιμοποιήσουμε τη συνάρτηση sqrt από τη math library. (το a^2 αντικαταστάθηκε από $a * a$, αντίστοιχα και τα υπόλοιπα τετράγωνα, διαφορετικά θα έπρεπε να χρησιμοποιήσουμε τη συνάρτηση pow της math library).

Χρόνος υπολογισμού της εικόνας: 8066 ms.

11.2. Βελτιστοποίηση ταυτόχρονου υπολογισμού

11.2.1. Threads

Όπως παρατηρούμε, κατά τη διάρκεια δημιουργίας της εικόνας, ο υπολογισμός της τιμής χρώματος κάθε pixel είναι εντελώς ανεξάρτητος από τα υπόλοιπα pixels. Αυτή η λεπτομέρεια κάνει τις εφαρμογές απεικόνισης fractals παρά πολύ δεκτικές στην υλοποίηση παραλληλισμού κατά την εκτέλεσή τους. Η εφαρμογή δημιουργεί threads σε ένα τετραγωνικό δισδιάστατο πλέγμα (grid), όπου το καθένα υπολογίζει μόνο το κομμάτι που του αντιστοιχεί. Τα threads μπορούν να ανατεθούν σε διαφορετικούς επεξεργαστές

δημιουργώντας την εικόνα παράλληλα, με αποτέλεσμα τη μείωση του χρόνου εκτέλεσης. Ο χρήστης δύναται να επιλέξει την πρώτη διάσταση του πλέγματος. Σε περίπτωση που επιλέξει το n σαν αριθμό, δημιουργείται ένα δισδιάστατο πλέγμα n x n από threads, όπου στο καθένα από τα threads ανατίθεται μια περιοχή της εικόνας.

Ακολουθεί η αλλαγή στον ψευδοκώδικα της drawImage() και της changePalette():

```
function drawImage()  {
    id = getThreadId();

    dim = grid_first_dimension;

    start_i = (id / dim) * (height / dim);
    end_i = (id  / dim + 1) * (height / dim);
    start_j = (id % dim) * (width / dim);
    end_j = (id % dim + 1) * (width / dim);

    for(i = start_i; i < end_i; i++) {
        for(j = start_j; j < end_j; j++) {
            complex = convertPixeltoComplex(i,  j);
            image_values[i][j] = iterativeFunction(complex);
            color = getColor(image_values[i][j], offset);
            paintPixel(image, i, j, color);
        }
    }

}
```

```
function changePalette()  {
    id = getThreadId();

    dim = grid_first_dimension;

    start_i = (id / dim) * (height / dim);
    end_i = (id  / dim + 1) * (height / dim);
    start_j = (id % dim) * (width / dim);
    end_j = (id % dim + 1) * (width / dim);

    for(i = start_i; i < end_i; i++) {
        for(j = start_j; j < end_j; j++) {
            color = getColor(image_values[i][j], offset);
            paintPixel(image, i, j, color);
        }
    }

}
```

Χρόνος υπολογισμού της εικόνας για n = 2, 2x2 πλέγμα, 4 threads: 4283 ms (Στο χρόνο συμπεριλαμβάνεται η βελτιστοποίηση norm squared).

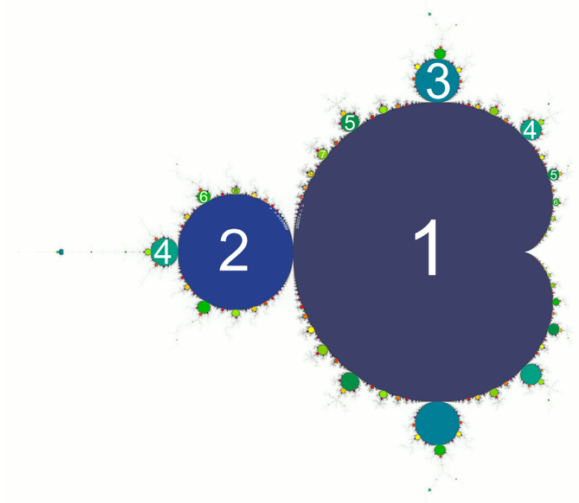
11.3. Βελτιστοποιήσεις των επαναληπτικών συναρτήσεων

11.3.1. Mandelbrot Optimization

Επειδή το Mandelbrot ήταν η πρώτη συνάρτηση που προστέθηκε στην εφαρμογή, ήταν και η πρώτη στην οποία έγινε προσπάθεια βελτιστοποίησης του χρόνου υπολογισμού της.

Όπως φαίνεται και από το σχήμα του, όλες οι μαύρες περιοχές ζωγραφίζονται πάντα στην ίδια προκαθορισμένη θέση. Οπότε, αν μπορούσαμε με κάποιο τρόπο να γνωρίζουμε εξ αρχής αν ένα pixel ανήκει στη μαύρη περιοχή, δε θα ήταν ανάγκη να εκτελέσουμε την επαναληπτική συνάρτηση για το μέγιστο αριθμό επαναλήψεων, δαπανώντας έτσι περισσότερο χρόνο.

Συγκεκριμένα, χρησιμοποιώντας την εξίσωση του κύκλου και την εξίσωση του καρδιοειδούς, μπορούμε απ ευθείας να γνωρίζουμε αν ένα pixel ανήκει στη μαύρη περιοχή ή όχι.



Η εφαρμογή πραγματοποιούσε το συγκεκριμένο έλεγχο στις περιοχές 1, 2 και 3 (όπου στο 3 εννοείται και ο συμμετρικός κύκλος).

Θα δείξουμε τις δυο εξισώσεις για τις περιοχές 1 και 2 (καρδιοειδές / κύκλος), αφού οι κύκλοι στο 3 έχουν απλά διαφορετικό κέντρο και διαφορετική ακτίνα.

Για το καρδιοειδές (1) έχουμε,

$$q = \left(x - \frac{1}{4}\right)^2 + y^2$$

$$q\left(q+\left(x-\frac{1}{4}\right)\right)<\frac{1}{4}y^2.$$

Αν ισχύει η ανισότητα, τότε το σημείο που εξετάζουμε ανήκει στο καρδιοειδές και η επαναληπτική συνάρτηση επιστρέφει αμέσως το μέγιστο αριθμό επαναλήψεων.

Για τον κύκλο (2) έχουμε,

$$(x+1)^2+y^2<\frac{1}{16}$$

Αν ισχύει η ανισότητα, τότε το σημείο που εξετάζουμε ανήκει στον κύκλο και η επαναληπτική συνάρτηση επιστρέφει αμέσως το μέγιστο αριθμό επαναλήψεων.

Η συγκεκριμένη βελτιστοποίηση, μολονότι είχε συμπεριληφθεί στις πρώτες εκδόσεις της εφαρμογής, πλέον δεν συμπεριλαμβάνεται, διότι είχε περιορισμένο αποτέλεσμα. Λειτουργούσε μόνο στο Mandelbrot set και μείωνε το χρόνο μόνο σε μικρά επίπεδα μεγέθυνσης, όσο δηλαδή υπήρχε στην εικόνα μας κάποιο κομμάτι περιοχής από τις προαναφερθείσες 1, 2 και 3. Φυσικά σε όλες τις υπόλοιπες περιπτώσεις απλά πρόσθετε στην εφαρμογή overhead λόγω των επιπλέον περιττών ελέγχων.

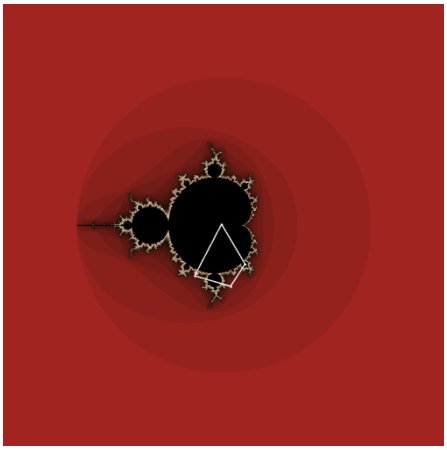
Γενικά, όταν γίνεται αναφορά σε περιοχές με μαύρο χρώμα, απλά εννοείται, ότι χρειάστηκαν να εκτελεστούν για το μέγιστο αριθμό επαναλήψεων. Όπως προαναφέρθηκε, ο χρήστης μπορεί να επιλέξει κάποιο άλλο χρώμα, διαφορετικό του μαύρου, για να αντιστοιχίσει αυτές τις περιοχές.

11.3.2. Periodicity Checking

Ο συγκεκριμένος αλγόριθμος προσπαθεί να λύσει το πρόβλημα, που δημιουργούσε ο προηγούμενος αλγόριθμος (λειτουργούσε μόνο για το Mandelbrot, είχε περιορισμένη εμβέλεια).

Βασίζεται στην παρατήρηση, ότι τα σημεία που ανήκουν στις μαύρες περιοχές ακολουθούν κάποια κυκλική ή περιοδική τροχιά.

Παρατηρήστε για παράδειγμα την παρακάτω τροχιά.



Αν η επαναληπτική συνάρτηση εισέλθει σε κάποιο αντίστοιχο κύκλο, είναι λογικό πως δεν θα ξεφύγει ποτέ από αυτόν, οπότε αν μέχρι τότε δεν έχει τερματίσει η επαναληπτική συνάρτηση από τους ελέγχους της νόρμας, είναι βέβαιο ότι θα πρέπει να εκτελεστεί για το μέγιστο αριθμό επαναλήψεων.

Θα δείξουμε συγκεκριμένα την αλλαγή του ψευδοκώδικα για τη συνάρτηση Mandelbrot, διότι ο ψευδοκώδικας για όλες τις άλλες συναρτήσεις αλλάζει με όμοιο τρόπο.

Η συγκεκριμένη βελτιστοποίηση δε χρησιμοποιείται στις Root Finding Methods και στη Sierpinski Gasket function, αφού έχουν ελάχιστες έως καθόλου μαύρες περιοχές και άρα η βελτιστοποίηση θα πρόσθετε overhead.

```
function double fractalFunction(Complex pixel) {
    pixel = applyRotation(pixel);
    pixel = applyTransformation(pixel);
    z = applyPerturbation(pixel);
    c = pixel;

    check = 3;
    check_counter = 0;
    update = 10;
    update_counter = 0;
    period = Complex(0, 0);

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(z.real * z .real + z.imaginary * z.imaginary >= bailout *
bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
        if(periodicityCheck(z)) {
            return max_iterations;
        }
    }
```

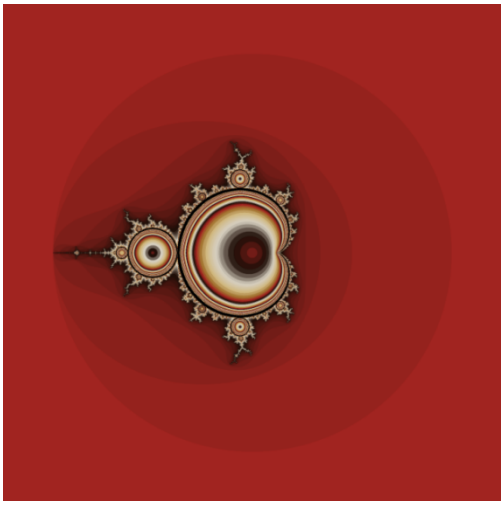
```
    return max_iterations;
}
function double juliaFunction(Complex pixel) {
    pixel = applyRotation(pixel);
    z = pixel;
    c = seed;

    check = 3;
    check_counter = 0;
    update = 10;
    update_counter = 0;
    period = Complex(0, 0);

    for(iterations = 0; iterations < max_iterations; iterations++) {
        if(z.real * z .real + z.imaginary * z.imaginary >= bailout *
bailout) {
            return colorAlgorithm(iterations,…,);
        }
        z = z^2 + c;
        if(periodicityCheck(z)) {
            return max_iterations;
        }
    }

    return max_iterations;
}
```

Όπου ο ψευδοκώδικας για την συνάρτηση periodicityCheck() είναι ο ακόλουθος:

```
function boolean periodicityCheck(Complex pixel) {

    if(|z - period| < error) {
        return true;
    }

    if(check == check_counter) {
        counter = 0;
        if(update == update_counter) {
            update_counter = 0;
            check = check * 2;
        }
        update_counter++;
        period = z;
    }

    check_counter++;
    return false;

}
```

Όπου error είναι κάποια πολύ μικρή τιμή (π.χ. 1e-13). Στον παραπάνω ψευδοκώδικα στον έλεγχο της νόρμας μπορούμε φυσικά να χρησιμοποιήσουμε τη βελτιστοποίηση norm squared, διότι χάριν απλότητας γράφθηκε με αυτό τον τρόπο.

Σχηματικά ο ψευδοκώδικας θα είχε το παρακάτω αποτέλεσμα:



εάν αντί του

```
if(periodicityCheck(z)) {
    return max_iterations;
}
```

είχαμε,

```
if(periodicityCheck(z)) {
    return iterations;
}
```

Χρόνος υπολογισμού της εικόνας: 779 ms (Στο χρόνο συμπεριλαμβάνεται η βελτιστοποίηση norm squared και threads).

Φυσικά ο συγκεκριμένος αλγόριθμος έχει πολύ πιο εμφανή αποτελέσματα, όταν ο μέγιστος αριθμός επαναλήψεων είναι αρκετά μεγάλος.

Όπως παρατηρούμε στην εικόνα ο αλγόριθμος δε λειτουργεί μόνο για τις περιοχές 1, 2 και 3 όπως ο προηγούμενος, αλλά για όλες τις μαύρες περιοχές. Επίσης λόγω της γενικότητάς του λειτουργεί και για τις υπόλοιπες επαναληπτικές συναρτήσεις.

11.4. Βελτιστοποιήσεις του αλγορίθμου ζωγραφίσματος

Η συγκεκριμένη ομάδα βελτιστοποιήσεων προσπαθεί με τη χρήση άπληστων αλγόριθμων, να μειώσει τον αριθμό των pixels που πρέπει να υπολογιστούν χρησιμοποιώντας τις επαναληπτικές συναρτήσεις.

Η λογική των άπληστων αλγόριθμων είναι η ακόλουθη. Δε θα υπάρξουν περιοχές που αποτελούνται από ένα χρώμα, μέσα σε περιοχές διαφορετικού χρώματος. Αυτό σημαίνει, πως αν υπολογίσουμε το περίγραμμα μιας περιοχής που αποτελείται από ένα χρώμα, τότε όλα τα pixels μέσα σε αυτή την περιοχή θα είναι του ιδίου χρώματος με το περίγραμμα. Με αυτό τον τρόπο όλα τα εσωτερικά pixels δε χρειάζονται να υπολογιστούν με το συμβατικό τρόπο.

Φυσικά οι συγκεκριμένοι αλγόριθμοι είναι δυνατόν να εισάγουν σφάλματα στην εικόνα, αλλά με την προσεκτική επιλογή των παραμέτρων τους μπορούμε να έχουμε πάρα πολύ καλό αποτέλεσμα, ενώ ο χρόνος εκτέλεσης μειώνεται αισθητά.

Τα περισσότερα σφάλματα στην εικόνα εμφανίζονται σε συναρτήσεις που πραγματοποιούν έλεγχο νόρμας με κριτήριο σύγκλισης, δηλαδή στις Magnet functions και στις Root Finding Methods.

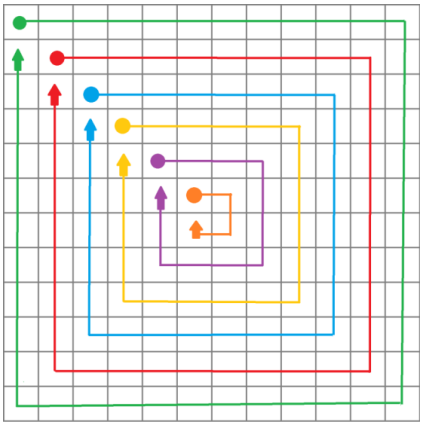
Σε αυτή την ομάδα βελτιστοποιήσεων δε θα παρατεθεί ψευδοκώδικας, καθόσον είναι πιο περίπλοκος. Γίνεται λοιπόν προσπάθεια με τη χρήση παραδειγμάτων ή με κάποια περιγραφή του αλγορίθμου να εξηγηθούν τα βασικά ζητήματα που προκύπτουν.

11.4.1. Solid Guessing

Ο συγκεκριμένος αλγόριθμος υπολογίζει όλα τα pixels του περιγράμματος μιας τετράγωνης (ορθογώνιας) περιοχής. Εάν όλα τα pixels του περιγράμματος είναι ίδιου χρώματος, τότε χρωματίζουμε τα pixels που περιέχονται σε αυτή την περιοχή με το χρώμα του περιγράμματος.

Υπάρχουν δύο υλοποιήσεις του συγκεκριμένου αλγόριθμου. Η μια τεχνική είναι να κινούμαστε με σπειροειδή τρόπο προς το κέντρο, ενώ με τη δεύτερη, εφαρμόζοντας την τεχνική «διαίρει και βασίλευε» (divide and conquer), διαιρούμε συνεχώς στο ήμισυ το αρχικό πρόβλημα.

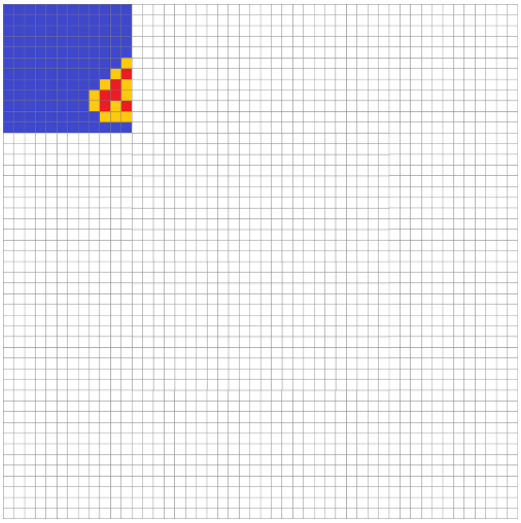
Αρχικά θα αναφέρουμε τη σπειροειδή τεχνική, το παρακάτω σχήμα δείχνει τα βήματά της,



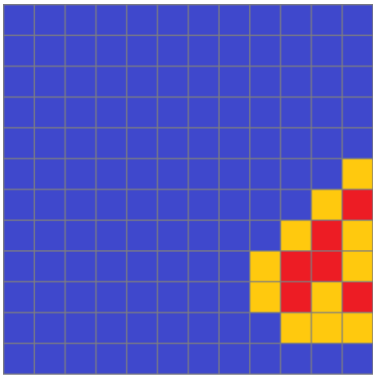
Στο πρώτο βήμα, υπολογίζουμε την εξωτερική περίμετρο της περιοχής που σκοπεύουμε να χρωματίσουμε. Αν κάθε pixel είναι του ίδιου χρώματος με το αρχικό pixel τότε ο αλγόριθμος τερματίζει και χρωματίζουμε τα pixels που απέμειναν με το χρώμα του αρχικού pixel. Διαφορετικά κινούμαστε ένα επίπεδο πιο μέσα και επαναλαμβάνουμε την ίδια διαδικασία.

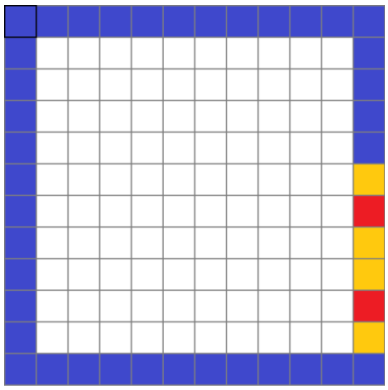
Φυσικά αυτή την τεχνική δεν την εφαρμόζουμε αμέσως σε όλη την εικόνα μας. Για να πετύχουμε καλύτερα αποτελέσματα χωρίζουμε την εικόνα σε μικρότερα κομμάτια και εφαρμόζουμε τον αλγόριθμο σε κάθε κομμάτι χωριστά.

Για παράδειγμα, χωρίζουμε την αρχική μας εικόνα σε υποπεριοχές των 12x12 pixels.

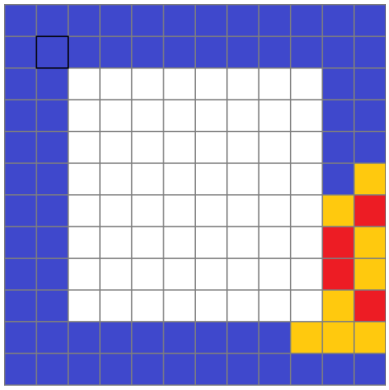


Ακολουθεί η εκτέλεση του αλγορίθμου για την $1^{η}$ υποπεριοχή.

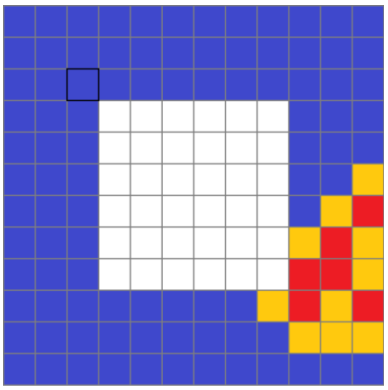




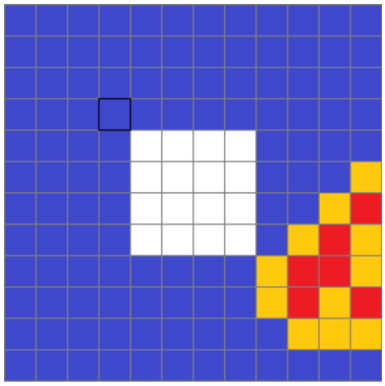
Βήμα 1



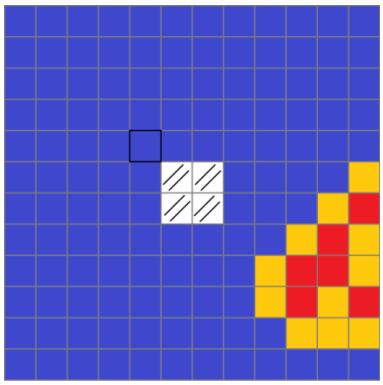
Βήμα 2



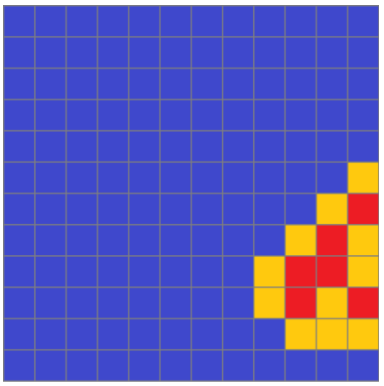
Βήμα 3



Βήμα 4



Βήμα 5

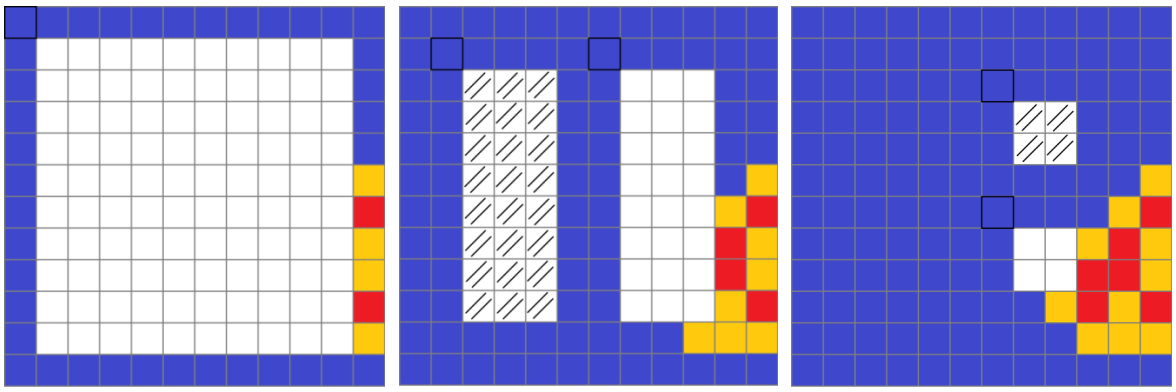


Βήμα 6

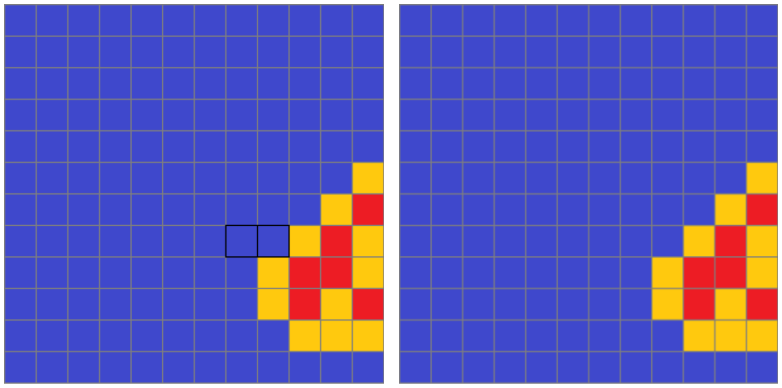
Στα βήματα 1 έως 4, δεν συνέβη τερματισμός της επανάληψης, καθόσον σε όλες τις περιμέτρους υπήρχαν pixels διαφορετικού χρώματος. Στο βήμα 5 παρατηρούμε όμως, ότι όλα τα pixels είναι του ιδίου χρώματος με το αρχικό pixel, οπότε μπορούμε να τερματίσουμε τον αλγόριθμο και να χρωματίσουμε αντίστοιχα το εσωτερικό εκείνης της περιοχής.

Η τεχνική «διαίρει και βασίλευε» υπολογίζει πάλι την εξωτερική περίμετρο με τον ίδιο τρόπο, δηλαδή αν κάθε pixel είναι του ιδίου χρώματος με το αρχικό pixel, τότε ο αλγόριθμος τερματίζει και χρωματίζουμε τα pixels που απέμειναν με το χρώμα του αρχικού pixel. Διαφορετικά η εικόνα διαιρείται σε δύο περιοχές και ο αλγόριθμος επαναλαμβάνεται κατά τον ίδιο τρόπο.

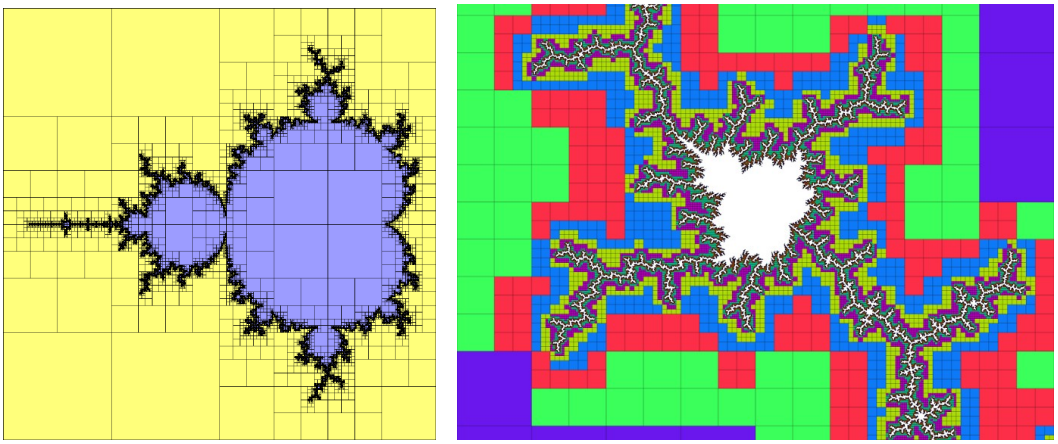
Ακολουθεί παράδειγμα εκτέλεσης του αλγορίθμου.



Βήμα 1 Βήμα 2 Βήμα 3

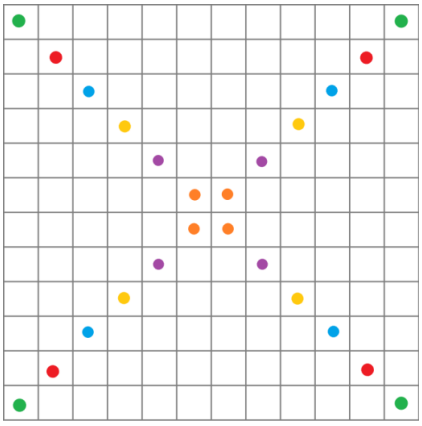


Βήμα 4 Βήμα 5

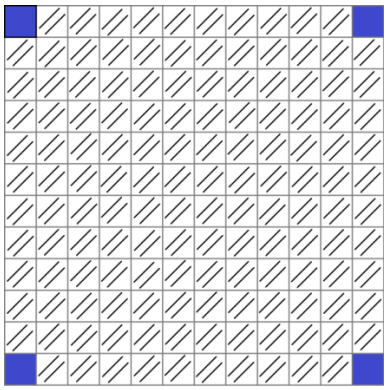


Παραδείγματα «διαίρει και βασίλευε»

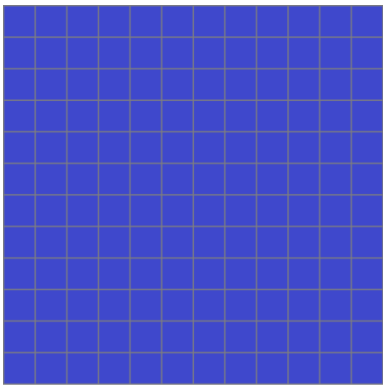
Μια ακόμα παραλλαγή που δύναται να εφαρμοστεί και στις δυο τεχνικές, είναι αντί να υπολογίζεται ολόκληρη η περίμετρος, υπολογίζονται μόνο οι γωνίες του τετραγώνου (ορθογωνίου) και αν είναι όλες του ίδιου χρώματος, τότε το εσωτερικό χρωματίζεται αντιστοίχως. Το παρακάτω σχήμα δείχνει τα βήματά της.



Η συγκεκριμένη παραλλαγή είναι πολύ άπληστη και μπορεί να εισάγει μεγαλύτερα σφάλματα, όπως για παράδειγμα:



Βήμα 1

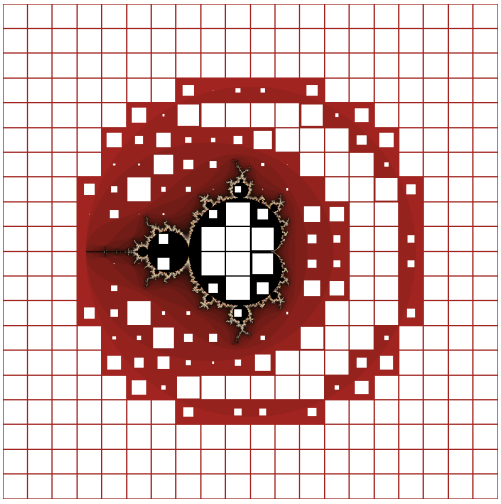


Βήμα 2

Η εφαρμογή, μέχρι την τελευταία έκδοση, χρησιμοποιούσε τη σπειροειδή solid guessing τεχνική, με υπολογισμό ολόκληρης της περιμέτρου.

Κάθε thread τεμάχιζε την περιοχή του σε ένα 10x10 πλέγμα και για κάθε κελί (cell) του πλέγματος εφήρμοζε το συγκεκριμένο αλγόριθμο.

Για 4 threads σε 2x2 πλέγμα, το αποτέλεσμα ήταν το ακόλουθο:



Όπου οι άσπρες περιοχές δε χρειάζεται να υπολογιστούν. Σε σχέση με τον κανονικό αλγόριθμο ζωγραφίσματος, μόνο το 28.48% των pixels υπολογίζεται με τη χρήση επαναληπτικής συνάρτησης.

Χρόνος υπολογισμού της εικόνας: 520 ms (Στο χρόνο συμπεριλαμβάνεται η βελτιστοποίηση norm squared, threads, periodicity checking).

11.4.2. Boundary Tracing

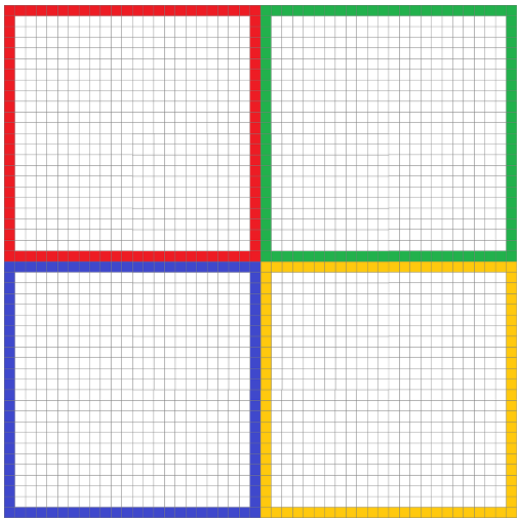
Ο συγκεκριμένος αλγόριθμος είναι ακόμη πιο σύνθετος σε σχέση με τον αλγόριθμο solid guessing. Η διαφορά του έγκειται στο γεγονός, ότι προσπαθεί να υπολογίσει όσο το δυνατόν τα λιγότερα pixels με χρήση επαναληπτικών συναρτήσεων. Για να το πετύχει αυτό, προσπαθεί να ακολουθήσει ολόκληρο το περίγραμμα μιας περιοχής, μέχρι να μην μπορεί πλέον να φτάσει πουθενά αλλού. Σε αυτό το σημείο, όποια περιοχή είναι ανάμεσα στο περίγραμμα ενός χρώματος, χρωματίζεται με το ίδιο χρώμα.

Εάν μία εικόνα περιέχει κάποια περιοχή χρώματος, αλλά ο αλγόριθμος δεν μπορεί να φτάσει σε αυτήν, τότε εσφαλμένα αυτή η περιοχή δεν θα συμπεριληφθεί στην εικόνα που δημιουργείται. Με αυτό τον τρόπο εισάγονται σφάλματα.

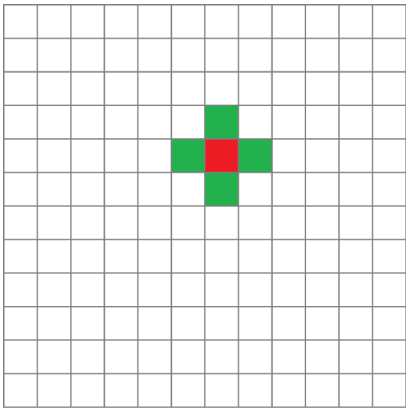
Το τρέξιμο του αλγορίθμου εξαρτάται αποκλειστικά από την εικόνα αυτή καθ αυτή, οπότε δεν είναι δυνατόν να παρουσιαστούν παραδείγματα εκτέλεσης. Θα αναλυθεί η λογική εκτέλεσης του αλγορίθμου.

Έστω ότι έχουμε 4 threads σε 2x2 πλέγμα.

Αρχικά κάθε thread προσθέτει σε μια λίστα επεξεργασίας (το κάθε thread έχει τη δική του λίστα) όλα τα pixels του περιγράμματος της περιοχής του, δηλαδή το $1^{ο}$ thread θα προσθέσει όλα τα κόκκινα pixels, το $2^{ο}$ τα πράσινα, το $3^{ο}$ τα μπλε και το $4^{ο}$ τα κίτρινα.



Από αυτό το σημείο κάθε thread εκτελεί τους ίδιους υπολογισμούς. Για κάθε ένα pixel της λίστας υπολογίζει την τιμή χρώματός του με χρήση των επαναληπτικών συναρτήσεων, καθώς και την τιμή των γειτονικών του pixels (πάνω, κάτω, αριστερά και δεξιά), φυσικά μόνο όταν τα γειτονικά pixels ανήκουν στο αντίστοιχο κομμάτι του thread.

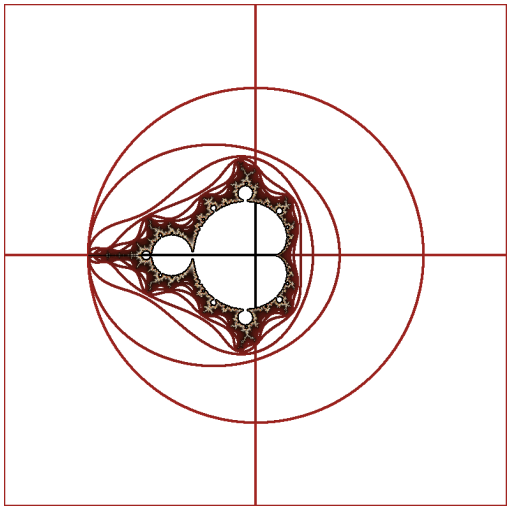


Κόκκινο: τρέχον pixel στη λίστα επεξεργασίας, Πράσινο: γειτονικά pixels

Αν κάποιο από τα γειτονικά pixels έχει διαφορετικό χρώμα από το τρέχον pixel της λίστας επεξεργασίας, τότε προστίθεται στη λίστα. Σε αυτό το βήμα το τρέχον pixel αφαιρείται από τη λίστα επεξεργασίας. Ο αλγόριθμος επαναλαμβάνεται για τα επόμενα pixels της λίστας επεξεργασίας, μέχρι να αδειάσει η λίστα.

Στο τέλος κάθε thread διατρέχει το αντίστοιχο υποκομμάτι της εικόνας, γραμμή-γραμμή και χρωματίζει τα pixels μεταξύ των περιγραμμάτων, χρησιμοποιώντας το αντίστοιχο χρώμα του περιγράμματος.

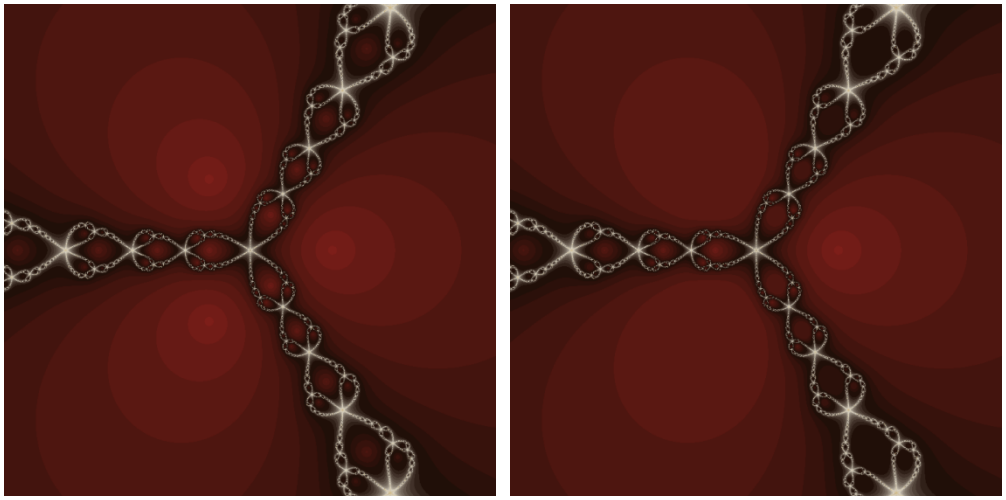
Για 4 threads σε 2x2 πλέγμα, το αποτέλεσμα είναι το ακόλουθο:



Όπου οι άσπρες περιοχές δε χρειάζεται να υπολογιστούν. Σε σχέση με τον κανονικό αλγόριθμο ζωγραφίσματος, μόνο το 5.03% των pixels υπολογίζεται με τη χρήση επαναληπτικής συνάρτησης.

Χρόνος υπολογισμού της εικόνας: 265 ms (Στο χρόνο συμπεριλαμβάνεται η βελτιστοποίηση norm squared, threads, periodicity checking).

Παράδειγμα προβληματικής συμπεριφοράς του αλγορίθμου:



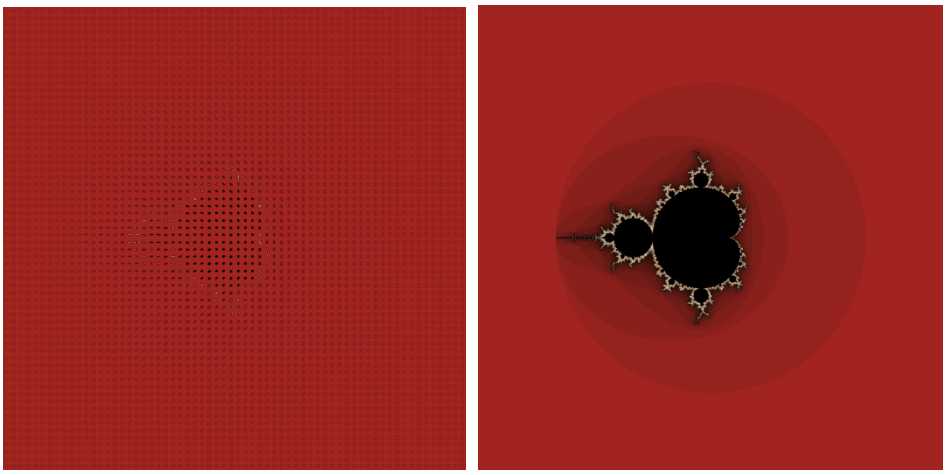
Κανονική εικόνα Boundary Tracing

12. Χάρτης των Julia sets (Julia Map)

Μέχρι στιγμής δεν έχει γίνει καμία αναφορά στη σχέση που υπάρχει στην κανονική επαναληπτική συνάρτηση (fractalFunction()) και στα Julia sets της (juliaFunction()). Το βέβαιο είναι ότι μοιράζονται τον ίδιο επαναληπτικό κανόνα και απλά έχουν διαφορετική τιμή στη μεταβλητή c.

Εξ ορισμού, αυτό που ισχύει είναι το εξής. Έστω ότι η τιμή του seed της juliaFunction() ισούται με seed = a + bi. Αν η τροχιά του (0, 0) (δηλαδή για z = 0 + 0i) μετά το πέρας της επαναληπτικής συνάρτησης (juliaFunction()) παραμένει φραγμένη, δηλαδή το χρώμα που αντιστοιχεί στο pixel του μιγαδικού αριθμού 0 + 0i είναι μαύρο, τότε το pixel που αντιστοιχεί στο μιγαδικό αριθμό a + bi (για τη συνάρτηση fractalFunction()) θα είναι επίσης μαύρο. Αν ο μιγαδικός αριθμός 0 + 0i δεν παραμείνει φραγμένος, τότε ούτε ο a + bi θα παραμείνει φραγμένος.

Ας αφήσουμε όμως τη θεωρία και ας το δούμε στην πράξη. Η επιλογή Julia Map, ζητά από το χρήστη να δώσει την πρώτη διάσταση ενός τετραγωνικού δισδιάστατου πλέγματος. Σε κάθε κελί του πλέγματος ανατίθεται ένα thread, το οποίο ζωγραφίζει ένα Julia set, χρησιμοποιώντας σαν seed το κέντρο του κελιού. Έχοντας σαν n = 64, δημιουργείται ένα πλέγμα 64x64 από threads, το οποίο έχει σαν αποτέλεσμα την ακόλουθη εικόνα:



Julia Map Κανονική εικόνα

Όπως παρατηρούμε, υπάρχει άμεση σχέση μεταξύ των δυο εικόνων και εδώ είναι τελικά το σημείο που αποκτά νόημα ο ορισμός. Εάν ο χρήστης δημιουργούσε ένα τετραγωνικό πλέγμα μεγέθους width x height (width = N, height = N), τότε σε κάθε thread θα ανατίθονταν ένα κελί που θα περιείχε ένα pixel. Κάθε thread θα έπρεπε να σχεδιάσει ένα Julia set χρησιμοποιώντας σαν seed το κεντρικό pixel του κελιού. Δεδομένου όμως ότι κάθε κελί έχει μόνο ένα pixel, αυτό θα ήταν και το seed (seed = a + bi). Οπότε, για να δημιουργήσει κάθε thread την αντίστοιχη εικόνα, θα έπρεπε να εκτελέσει την συνάρτηση

juliaFunction() για όλα τα pixels του κελιού του. Δεδομένου όμως, ότι το κελί έχει μόνο ένα pixel, θα είχαμε z = 0 + 0i (μετά από χρήση της συνάρτησης convertPixelToComplex()), γεγονός που αποδεικνύει και τον ορισμό. Αν το πλέγμα ήταν width x height, τότε οι δύο παραπάνω εικόνες θα ήταν ίδιες.

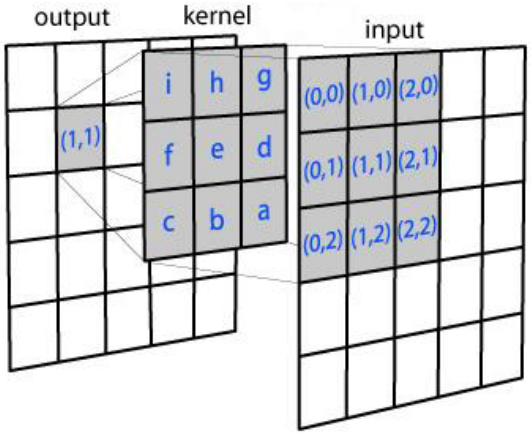
13. Φίλτρα εικόνας (Filters)

Έχοντας πλέον δημιουργήσει μια εικόνα, μπορούμε χρησιμοποιώντας ένα από τα φίλτρα να εμφανίσουμε ένα διαφορετικό αισθητικό αποτέλεσμα.

Το φίλτρο είναι ουσιαστικά μια συνάρτηση, η οποία παίρνει σαν είσοδο μια εικόνα και μας επιστρέφει σαν αποτέλεσμα μια διαφορετική εικόνα, η οποία έχει υποστεί κάποιας μορφής επεξεργασία.

Η επεξεργασία εικόνας, αυτή καθ αυτή, είναι από μονή της πολύ μεγάλο κεφάλαιο συζήτησης, οπότε εδώ δε θα μπούμε σε αρκετές λεπτομέρειες. Ένα από τα σημαντικότερα εργαλεία της επεξεργασίας εικόνας είναι η συνέλιξη.

Έχοντας μια εικόνα είσοδο (input, όπου κάθε pixel της αποτελείται από τις τιμές χρώματος red, green και blue), και ένα πινάκα kernel με συντελεστές, τότε σύμφωνα με το παρακάτω σχήμα, το χρώμα του pixel (1, 1) θα είναι:



$$[1,1].r = i * [0,0].r + h * [1,0].r + g * [2,0].r + f * [0,1].r + e * [1,1].r + d * [2,1].r + c * [0,2].r + b * [1,2].r + a * [2,2].r$$

$$[1,1].g = i * [0,0].g + h * [1,0].g + g * [2,0].g + f * [0,1].g + e * [1,1].g + d * [2,1].g + c * [0,2].g + b * [1,2].g + a * [2,2].g$$

$$[1,1].b = i * [0,0].b + h * [1,0].b + g * [2,0].b + f * [0,1].b + e * [1,1].b + d * [2,1].b + c * [0,2].b + b * [1,2].b + a * [2,2].b$$

Όπου r, g, b είναι οι τιμές των red, green και blue αντίστοιχα. Ακολουθώντας την ίδια διαδικασία για όλα τα pixels της εικόνας, δηλαδή πολλαπλασιάζοντας τον kernel με τα αντίστοιχα pixels και προσθέτοντας το συνολικό αποτέλεσμα, δημιουργείται η τελική εικόνα (output).

Οπτικά, ο αλγόριθμος λειτουργεί ως εξής. Κάθε φορά το κεντρικό pixel του kernel (το e στην περίπτωση μας) πρέπει να είναι πάνω στη θέση του pixel, που θέλουμε να υπολογίσουμε. Έχοντας αυτό σαν οδηγό, ξέρουμε ποιά pixels θα πρέπει να πολλαπλασιαστούν μεταξύ τους.

Πολλά από τα φίλτρα που θα αναλυθούν χρησιμοποιούν συνέλιξη για να παράγουν την εικόνα αποτέλεσμα. Απλά πρέπει να προσδιοριστεί ο πινάκας kernel, ώστε να έχουμε το επιθυμητό αποτέλεσμα. Η Java παρέχει μέθοδο που υλοποιεί τη συνέλιξη και χρειάζεται σαν είσοδο την εικόνα input, την εικόνα output και τον kernel.

Όλα τα φίλτρα εικόνας, εκτός του Anti-Aliasing (Supersampling), εφαρμόζονται στο τέλος της συνάρτησης drawImage() και αντίστοιχα στο τέλος της changePalette(), δηλαδή αφού έχει δημιουργηθεί η βασική εικόνα. Ακολουθεί η αλλαγή στον ψευδοκώδικά τους:

```
function drawImage()  {
    id = getThreadId();

    dim = grid_first_dimension;

    start_i = (id / dim) * (height / dim);
    end_i = (id  / dim + 1) * (height / dim);
    start_j = (id % dim) * (width / dim);
    end_j = (id % dim + 1) * (width / dim);

    for(i = start_i; i < end_i; i++) {
        for(j = start_j; j < end_j; j++) {
            complex = convertPixeltoComplex(i,  j);
            image_values[i][j] = iterativeFunction(complex);
            color = getColor(image_values[i][j], offset);
            paintPixel(image, i, j, color);
        }
    }

    if(id == 0 && filter == true) {
        applyFilter();
    }

}
```

```
function changePalette()  {
    id = getThreadId();

    dim = grid_first_dimension;

    start_i = (id / dim) * (height / dim);
    end_i = (id  / dim + 1) * (height / dim);
```

```
    start_j = (id % dim) * (width / dim);
    end_j = (id % dim + 1) * (width / dim);

    for(i = start_i; i < end_i; i++) {
        for(j = start_j; j < end_j; j++) {
            color = getColor(image_values[i][j], offset);
            paintPixel(image, i, j, color);
        }
    }

    if(id == 0 && filter == true) {
        applyFilter();
    }

}
```

Η συνάρτηση applyFilter() μπορεί να είναι μια από τα ακόλουθα φίλτρα,

- Anti-Aliasing (Blurring)
- Edge Detection
- Edge Detection 2
- Sharpness
- Emboss
- Emboss Colored
- Invert Colors

ή κάποιος συνδυασμός τους.

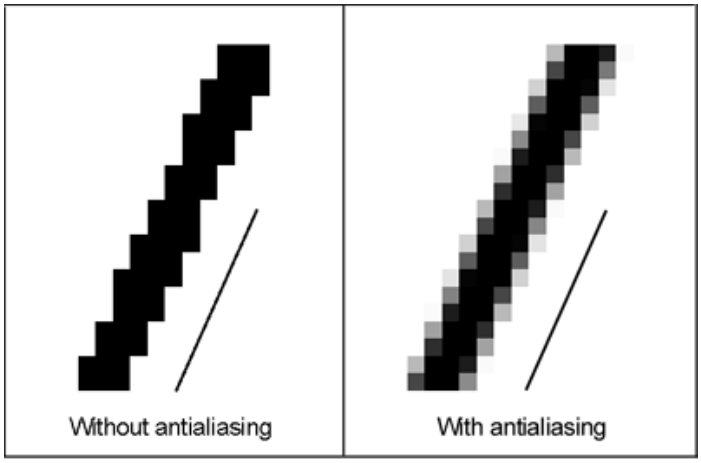
Ακολουθεί ο ψευδοκώδικας που υλοποιεί τη συνέλιξη σε μια εικόνα:

```
function convolve(image_in, image_out, kernel, kernel_first_dimension) {

    size = kernel_first_dimension;
    kernel_2 = size / 2;

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            sum_r = 0;
            sum_g = 0;
            sum_b = 0;
            for(k = i - kernel_2, p = 0; p < size; k++, p++) {
                if(k >= 0 && k < height) {
                    for(l = j - kernel_2, t = 0; t < size, l++, t++) {
                        if(l >= 0 && l < width) {
                            sum_r += image_in[k][l].red * kernel[p][t];
                            sum_g += image_in[k][l].green * kernel[p][t];
                            sum_b += image_in[k][l].blue * kernel[p][t];
                        }
                    }
                }
            }
            image_in[i][j].red = sum_r;
            image_in[i][j].green = sum_g;
            image_in[i][j].blue = sum_b;
        }
    }

}
```

13.1. Αντι-Ταύτιση (Anti-Aliasing)

Όλα τα θεμελιακά στοιχεία των γραφικών έχουν μια οδοντωτή εμφάνιση ή εμφάνιση σκαλοπατιών, διότι η διαδικασία δειγματοληψίας ψηφιοποιεί σημεία συντεταγμένων ενός αντικειμένου σε διακριτές ακέραιες τιμές pixels. Αυτή η παραμόρφωση της πληροφορίας λόγω της δειγματοληψίας χαμηλής συχνότητας (υποδειγματοληψία / ανεπαρκής δειγματοληψία – undersampling) ονομάζεται ταύτιση (aliasing).

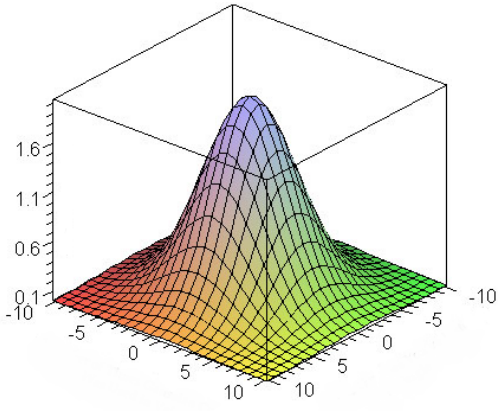
Μπορούμε να βελτιώσουμε την εμφάνιση της εικόνας, εφαρμόζοντας μεθόδους εξομάλυνσης ή αντι-ταύτισης ορίων (anti-aliasing) που αναπληρώνουν για τη διαδικασία υποδειγματοληψίας.



Παράδειγμα anti-aliasing σε γραμμή.

13.1.1. Αντι-Ταύτιση με χρήση blurring kernel

Μέχρι την τελευταία έκδοση της εφαρμογής, χρησιμοποιήθηκε ένα φίλτρο εικόνας, όμοιο περίπου με το Gaussian Filter, το οποίο πραγματοποιούσε μιας μορφής θόλωμα (blurring) στην εικόνα.



Gaussian Blur Filter

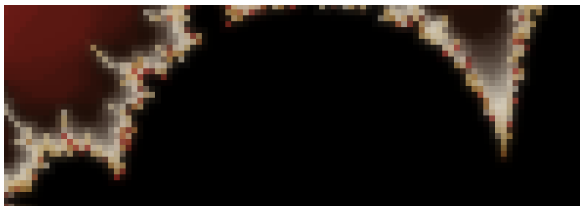
Για τη διαδικασία χρησιμοποιήθηκε συνέλιξη, ενώ ο kernel βασίζονταν στον παρακάτω πινάκα:

$$\begin{bmatrix} 0.00390625 & 0.00390625 & 0.00390625 & 0.00390625 & 0.00390625 \\ 0.00390625 & 0.04687500 & 0.04687500 & 0.04687500 & 0.00390625 \\ 0.00390625 & 0.04687500 & 0.56250000 & 0.04687500 & 0.00390625 \\ 0.00390625 & 0.04687500 & 0.04687500 & 0.04687500 & 0.00390625 \\ 0.00390625 & 0.00390625 & 0.00390625 & 0.00390625 & 0.00390625 \end{bmatrix}$$

Τα στοιχεία ολόκληρου του πινάκα αθροίζονται στο 1.

Αποτέλεσμα (έχοντας μεγεθύνει την εικόνα):

Αρχική Εικόνα



Anti-Aliasing (Blurring)

13.1.2. Αντι-Ταύτιση με υπερδειγματοληψία (supersampling)

Το πρόβλημα της παραπάνω τεχνικής είναι, ότι ναι μεν η νέα μας εικόνα έχει εξομαλυνθεί, κρατώντας όμως κάποιο ποσοστό χρώματος από κάθε pixel, ουσιαστικά έχουμε χάσει αρκετή πληροφορία, γι' αυτό το λόγο δημιουργείται και το θόλωμα.

Μια άλλη μέθοδος αντι-ταύτισης είναι να αυξήσουμε το ρυθμό δειγματοληψίας, χειριζόμενοι την εικόνα σαν να ήταν καλυμμένη από λεπτότερο πλέγμα, από ότι είναι στην πραγματικότητα. Τότε, μπορούμε να χρησιμοποιήσουμε πολλαπλά σημεία δειγμάτων στο

εύρος αυτού του λεπτότερου πλέγματος, ώστε να προσδιορίσουμε ένα κατάλληλο επίπεδο έντασης για κάθε pixel της εικόνας.

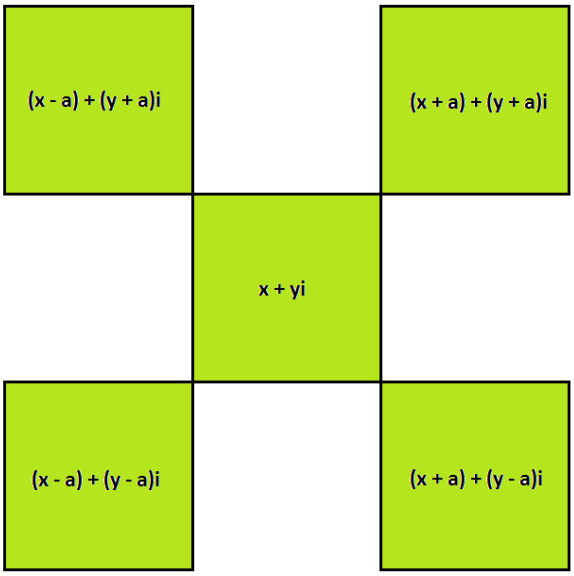
Αυτή η τεχνική δειγματοληψίας των χαρακτηριστικών των αντικειμένων σε υψηλότερη ανάλυση και η προβολή των αποτελεσμάτων σε χαμηλότερη ανάλυση ονομάζεται υπερδειγματοληψία (supersampling).

Ο απλός αλγόριθμος ζωγραφίσματος της εικόνας (drawImage()), μεταβάλλεται ως εξής:

```
function drawImage()  {
    id = getThreadId();

    dim = grid_first_dimension;


    start_i = (id / dim) * (height / dim);
    end_i = (id  / dim + 1) * (height / dim);
    start_j = (id % dim) * (width / dim);
    end_j = (id % dim + 1) * (width / dim);

    a = (size / N)* 0.25;

    for(i = start_i; i < end_i; i++) {
        for(j = start_j; j < end_j; j++) {
            complex = convertPixeltoComplex(i,  j);
            image_values[i][j] = iterativeFunction(complex);
            color = getColor(image_values[i][j], offset);

            value2 = iterativeFunction(complex + Complex(-a,-a));
            value3 = iterativeFunction(complex + Complex(a, -a));
            value4 = iterativeFunction(complex + Complex(a, a));
            value5 = iterativeFunction(complex + Complex(-a, a));

            color2 = getColor(value2, offset);
            color3 = getColor(value3, offset);
            color4 = getColor(value4, offset);
            color5 = getColor(value5, offset);

            red = (color.red + color2.red + color3.red + color4.red +
color5.red) / 5;
            green = (color. green + color2. green + color3. green +
color4. green + color5. green) / 5;
            blue = (color. blue + color2. blue + color3. blue +
color4. blue + color5. blue) / 5;

            color = Color(red, green, blue);
            paintPixel(image, i, j, color);
        }
    }

    if(id == 0 && filter == true) {
        applyFilter();
    }

}
```

Αντίστοιχα μεταβάλλεται και ο αλγόριθμος ζωγραφίσματος με boundary tracing.

Ο απλός αλγόριθμος ζωγραφίσματος, έπρεπε να υπολογίσει ένα pixel, έστω το x + yi, ενώ ο αλγόριθμος του oversampling πρέπει να υπολογίσει περισσότερα δείγματα, βάσει του παρακάτω σχήματος:



Όπου $a = \frac{size}{N} * 0.25$ (width = N, height = N).

Οι 4 νέοι μιγαδικοί αριθμοί δεν ανήκουν κανονικά στα δείγματα της αρχικής εικόνας, αλλά σε λεπτότερο πλέγμα.

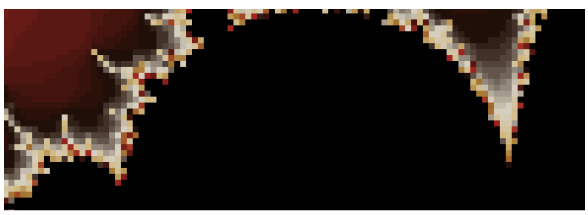
Στο τέλος για να παράγουμε το τελικό χρώμα, παίρνουμε το μέσο όρο των 5 χρωμάτων.

Pixel with sample postions

Resulting color

$$\frac{\square + \square + \square + \square}{4} = \square$$

Αποτέλεσμα (έχοντας μεγεθύνει την εικόνα):



Αρχική Εικόνα

Anti-Aliasing (Supersampling)

Στην εφαρμογή χρησιμοποιείται ο αλγόριθμος του supersampling.

13.2. Ανίχνευση ορίων (Edge Detection)

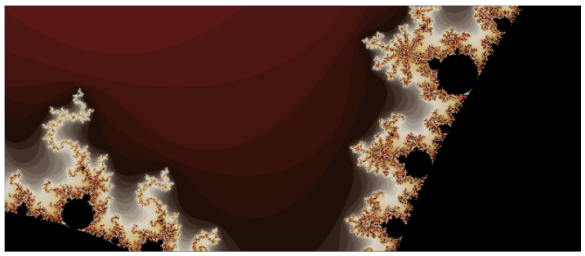
Ο συγκεκριμένος αλγόριθμος, εν αντιθέσει με τους αλγορίθμους αντι-ταύτισης που προσπαθούν να εξομαλύνουν την εικόνα, προσπαθεί να διατηρήσει μόνο τις «ανώμαλες» περιοχές της εικόνας.

13.2.1. Edge Detection

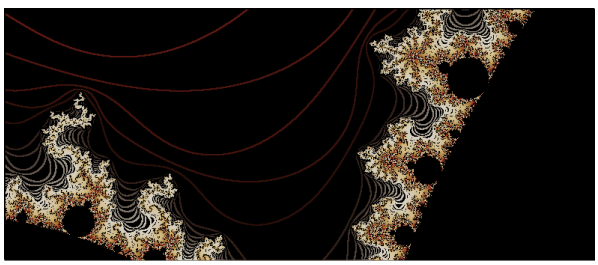
Χρησιμοποιούμε ένα kernel για συνέλιξη της εικόνας με σκοπό να πετύχουμε το επιθυμητό αποτέλεσμα. Η εικόνα παράγεται με τη χρήση χοντρών γραμμών. Ο kernel βασίζεται στον παρακάτω πινάκα:

$$\begin{bmatrix} -1.0 & -1.0 & -1.0 & -1.0 & -1.0 \\ -1.0 & -2.0 & -2.0 & -2.0 & -1.0 \\ -1.0 & -2.0 & 32.0 & -2.0 & -1.0 \\ -1.0 & -2.0 & -2.0 & -2.0 & -1.0 \\ -1.0 & -1.0 & -1.0 & -1.0 & -1.0 \end{bmatrix}$$

Αποτέλεσμα:



Αρχική Εικόνα



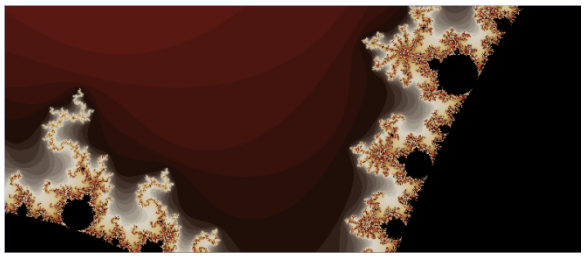
Edge Detection

13.2.2. Edge Detection 2

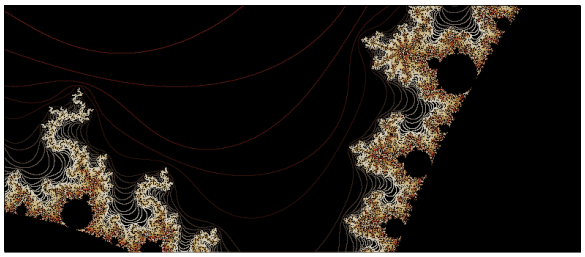
Χρησιμοποιούμε ένα kernel για συνέλιξη της εικόνας με σκοπό να πετύχουμε το επιθυμητό αποτέλεσμα. Η εικόνα παράγεται με τη χρήση λεπτών γραμμών. Ο kernel βασίζεται στον παρακάτω πινάκα:

$$\begin{bmatrix} -1.0 & -1.0 & -1.0 \\ -1.0 & 8.0 & -1.0 \\ -1.0 & -1.0 & -1.0 \end{bmatrix}$$

Αποτέλεσμα:



Αρχική Εικόνα



Edge Detection 2

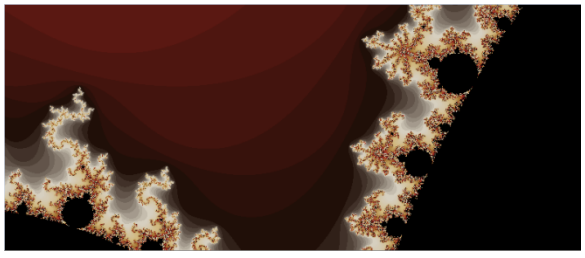
13.3. Όξυνση (Sharpness)

Ο συγκεκριμένος αλγόριθμος, εν αντιθέσει με τον αλγόριθμο ανίχνευσης ορίων διατηρεί όλες τις περιοχές της εικόνας, με τη διαφορά ότι οξύνει μόνο την ένταση των «ανώμαλων» περιοχών της εικόνας.

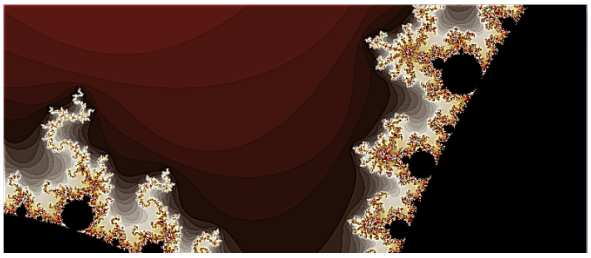
Χρησιμοποιούμε ένα kernel για συνέλιξη της εικόνας με σκοπό να πετύχουμε το επιθυμητό αποτέλεσμα. Ο kernel βασίζεται στον παρακάτω πινάκα:

$$\begin{bmatrix} -0.1 & -0.1 & -0.1 & -0.1 & -0.1 \\ -0.1 & -0.1 & -0.1 & -0.1 & -0.1 \\ -0.1 & -0.1 & 3.40 & -0.1 & -0.1 \\ -0.1 & -0.1 & -0.1 & -0.1 & -0.1 \\ -0.1 & -0.1 & -0.1 & -0.1 & -0.1 \end{bmatrix}$$

Αποτέλεσμα:



Αρχική Εικόνα



Sharpness

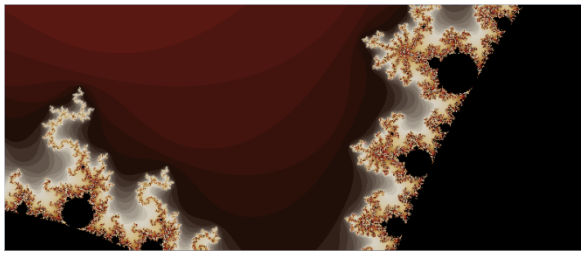
13.4. Emboss

Ο συγκεκριμένος αλγόριθμος δημιουργεί μια ψευδό-τρισδιάστατη απεικόνιση, αφού περιοχές διαφορετικού χρώματος φαίνονται να έχουν διαφορετικό ύψος/βάθος. Η εικόνα μετατρέπεται σε gray scale.

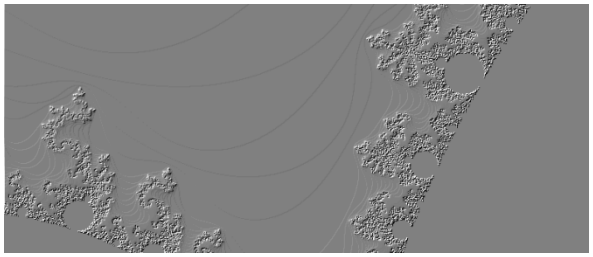
Κατ εξαίρεση θα ακολουθήσει ο κώδικας που υλοποιεί τον αλγόριθμο σε Java, διότι και βρέθηκε αυτούσιος στο διαδίκτυο:

```
private void filterEmboss() {

    int image_size = image.getHeight();

    BufferedImage newSource = new BufferedImage (image_size, image_size,
BufferedImage.TYPE_INT_RGB);

    for (int i = 0; i < image_size; i++) {
        for (int j = 0; j < image_size; j++) {
            int current = image.getRGB (j, i);

            int upperLeft = 0;
            if(i > 0 && j > 0) {
                upperLeft = image.getRGB (j - 1, i - 1);
            }

            int rDiff = ((current >> 16) & 255) - ((upperLeft >> 16) &
255);
            int gDiff = ((current >> 8) & 255) - ((upperLeft >> 8) & 255);
            int bDiff = (current & 255) - (upperLeft & 255);

            int diff = rDiff;
            if (Math.abs (gDiff) > Math.abs (diff)) {
                diff = gDiff;
            }
            if (Math.abs (bDiff) > Math.abs (diff)) {
                diff = bDiff;
            }

            int grayLevel = Math.max(Math.min (128 + diff, 255), 0);
            rgbs[i * image_size + j] = (grayLevel << 16) + (grayLevel << 8)
+ grayLevel;

        }
    }
    newSource.setRGB(0, 0, image_size, image_size, rgbs, 0, image_size);


    image.getGraphics().drawImage(newSource, 0, 0, image_size , image_size,
null);

    newSource = null;

}
```

Αποτέλεσμα:



Αρχική Εικόνα



Emboss

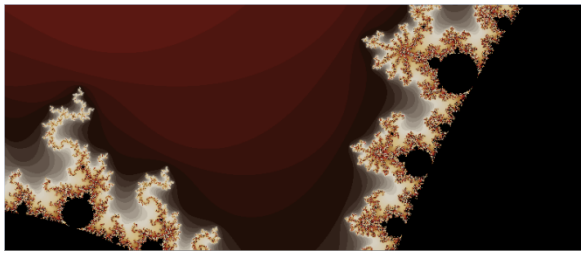
13.5. Emboss Colored

Ο συγκεκριμένος αλγόριθμος δημιουργεί μια ψευδό-τρισδιάστατη απεικόνιση, αφού περιοχές διαφορετικού χρώματος φαίνονται να έχουν διαφορετικό ύψος/βάθος. Η εικόνα διατηρεί τα αρχικά της χρώματα.

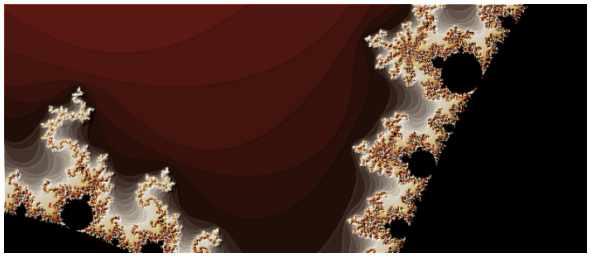
Χρησιμοποιούμε ένα kernel για συνέλιξη της εικόνας με σκοπό να πετύχουμε το επιθυμητό αποτέλεσμα. Ο kernel βασίζεται στον παρακάτω πινάκα:

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & -1.0 \end{bmatrix}$$

Αποτέλεσμα:



Αρχική Εικόνα



Emboss Colored

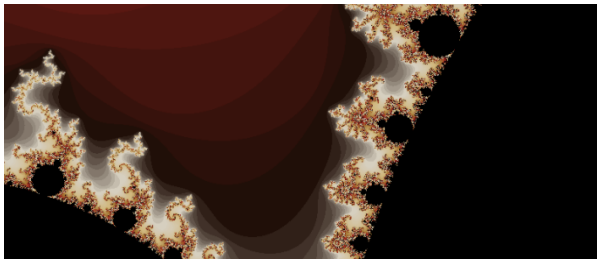
13.6. Αντίθετα Χρώματα (Inverted Colors)

Ο συγκεκριμένος αλγόριθμος μετατρέπει κάθε χρώμα της εικόνας στο αντίθετο (συμπληρωματικό) του χρώμα (δεδομένου ότι στο rgb format τα red, green και blue παίρνουν τιμές από 0 έως 255).

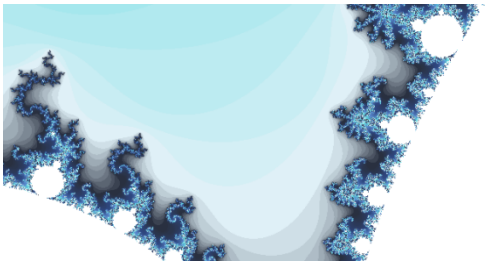
Ακολουθεί ο ψευδοκώδικας για τον αλγόριθμο invertColors():

```
function inverColors(image_in, image_out)  {

    for(i = 0; i < height; i++) {
        for(j = 0; j < width; j++) {
            image_out[i][j].red = 255 - image_in[i][j].red;
            image_out[i][j].green = 255 - image_in[i][j].green;
            image_out[i][j].blue = 255 - image_in[i][j].blue;
        }
    }

}
```

Αποτέλεσμα:



Αρχική Εικόνα



Inverted Colors

14. Επιπλέον λειτουργίες

Ο χρήστης:

1. μπορεί να αποθηκεύσει τις ρυθμίσεις από την επιλογή Save as και να τις ξαναφορτώσει με την επιλογή Load.

2. με την επιλογή Save Image As, μπορεί να αποθηκεύσει μια εικόνα σε ένα από τα ακόλουθα formats,
    - bmp
    - png
    - jpeg/jpg

3. με την επιλογή Starting Position, μπορεί να επαναφέρει το fractal στην αρχική (default) του θέση.

4. με την επιλογή Go To, μπορεί να επιλέξει ως κέντρο, όποιο μιγαδικό αριθμό θέλει με ακρίβεια. Επίσης μπορεί να επιλέξει κατ αυτόν τον τρόπο και το Julia set seed.

5. με την επιλογή Image Size, μπορεί να καθορίσει το μέγεθος της εικόνας που δημιουργείται (πάντοτε τετραγωνική).

6. με τις επιλογές Set Iterations και Bailout, μπορεί να καθορίσει το μέγιστο αριθμό επαναλήψεων και την τιμή του bailout αντίστοιχα.

7. με την επιλογή Show Grid, μπορεί να εμφανίσει ένα δισδιάστατο πλέγμα στην εικόνα, με αριθμούς μέτρησης.

8. τοποθετώντας τον δείκτη του ποντικιού πάνω στη μπάρα με το ποσοστό, μπορεί να βλέπει το χρόνο υπολογισμού της τελευταίας λειτουργίας, καθώς και το ποσοστό των pixels που υπολογιστήκαν με τη χρήση επαναληπτικών συναρτήσεων.

15. Σύνοψη – Συμπεράσματα

Είναι γεγονός, πως ο βαθμός παραμετροποίησης τέτοιου είδους εφαρμογών είναι εξαιρετικά υψηλός, με αποτέλεσμα να μην είναι δυνατόν να παραχθούν όλα τα πιθανά αποτελέσματα της εφαρμογής. Εξ' αυτού του λόγου και η έκφραση «κορυφή του παγόβουνου» που αναφέρθηκε στην εισαγωγή. Εκτός αυτού, το διαδίκτυο παρέχει αρκετά μεγάλο αριθμό, αντίστοιχων λογισμικών, το καθένα από τα οποία υλοποιεί μεγάλο αριθμό συναρτήσεων και επιλογών.

Με λίγα λόγια, το περιθώριο περαιτέρω ανάπτυξης της εφαρμογής είναι αρκετά μεγάλο, τόσο από άποψη περισσότερων λειτουργιών, όσο και στον τομέα της απόδοσης (φυσικά σε κάποια καλύτερη γλώσσα προγραμματισμού).

Έχοντας πλέον μια εμπεριστατωμένη άποψη αντίστοιχων εφαρμογών, που έχουν γραφθεί σε Java (μιας και η συγγραφή του κώδικα διήρκησε σχεδόν δύο χρόνια), η συγκεκριμένη εφαρμογή, αν δεν είναι η πιο περιεκτική από θέμα λειτουργιών και αρκετά γρήγορη (απόδοση) σε σχέση με παρόμοιες εφαρμογές γραμμένες σε Java, αναμφίβολα κατέχει εκ του αποτελέσματος υψηλή κατά την άποψή μου θέση.

16. Βιβλιογραφία

1. Γενικές πληροφορίες σχετικά με τα fractals, http://en.wikipedia.org/wiki/Fractal
2. Mandelbrot set, http://en.wikipedia.org/wiki/Mandelbrot_set
3. Multibrot set, http://en.wikipedia.org/wiki/Multibrot_set
4. Burning Ship, http://en.wikipedia.org/wiki/Burning_Ship_fractal
5. Magnet fractals, http://www.icd.com/tsd/fractals/beginner4.htm
6. Newton method fractals, http://en.wikipedia.org/wiki/Newton_fractal
7. Newton method fractals, http://www.chiark.greenend.org.uk/~sgtatham/newton/
8. Plane transformations, http://aleph0.clarku.edu/~djoyce/julia/altplane.html
9. Julia sets, http://en.wikipedia.org/wiki/Julia_set
10. Forums για fractals, http://www.fractalforums.com
11. FractInt, fractal software, http://www.nahee.com/spanky/www/fractint/
12. XaoS, fractal software, http://wmi.math.u-szeged.hu/xaos/doku.php
13. Fractal Extreme, fractal software, http://www.cygnus-software.com/
14. Boundary Tracing & fractal applets, http://www.ibiblio.org/e-notes/MSet/BigM.htm
15. Optimizations, Solid Guessing, Boundary Tracing, http://mrob.com/pub/muency.html
16. Boundary Tracing, http://bisqwit.iki.fi/jutut/kuvat/programming_examples/mandelbrotbtrace.pdf
17. Periodicity Checking, http://en.wikipedia.org/wiki/User:Simpsons_contributor/periodicity_checking
18. Mandelbrot software σε Java, http://math.hws.edu/xJava/MB/
19. Κώδικας του Mandelbrot, με χρήση antialiasing σε Java, http://java.rubikscube.info/
20. Κώδικες για το Mandelbrot set, με χρήση βελτιστοποιήσεων και αλγορίθμων χρωματισμού, http://en.wikibooks.org/wiki/Fractals/Iterations_in_the_complex_plane/Mandelbrot_set
21. Τεχνικές πληροφορίες για fractals και antialiasing, http://www.hpdz.net/
22. Antialiasing, http://en.wikipedia.org/wiki/Supersample_anti-aliasing
23. Edge Detection, http://en.wikipedia.org/wiki/Edge_detection
24. Embossing, http://en.wikipedia.org/wiki/Image_embossing
25. COLORING DYNAMICAL SYSTEMS IN THE COMPLEX PLANE: Francisco Garcia, Angel Fernandez, Javier Barrallo, Luis Martin, http://math.unipa.it/~grim/Jbarrallo.PDF
26. Γραφικά Υπολογιστών με OpenGL: Hearn Baker