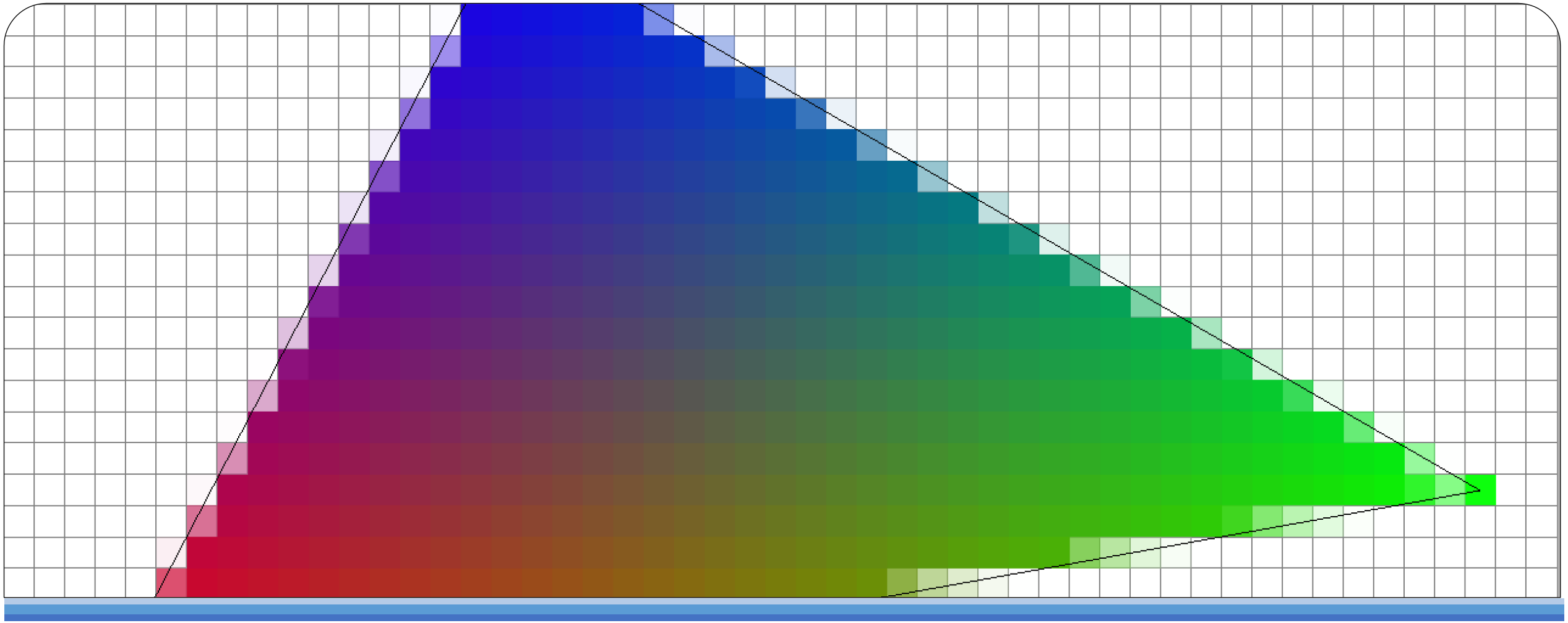


Γραφικά Υπολογιστών

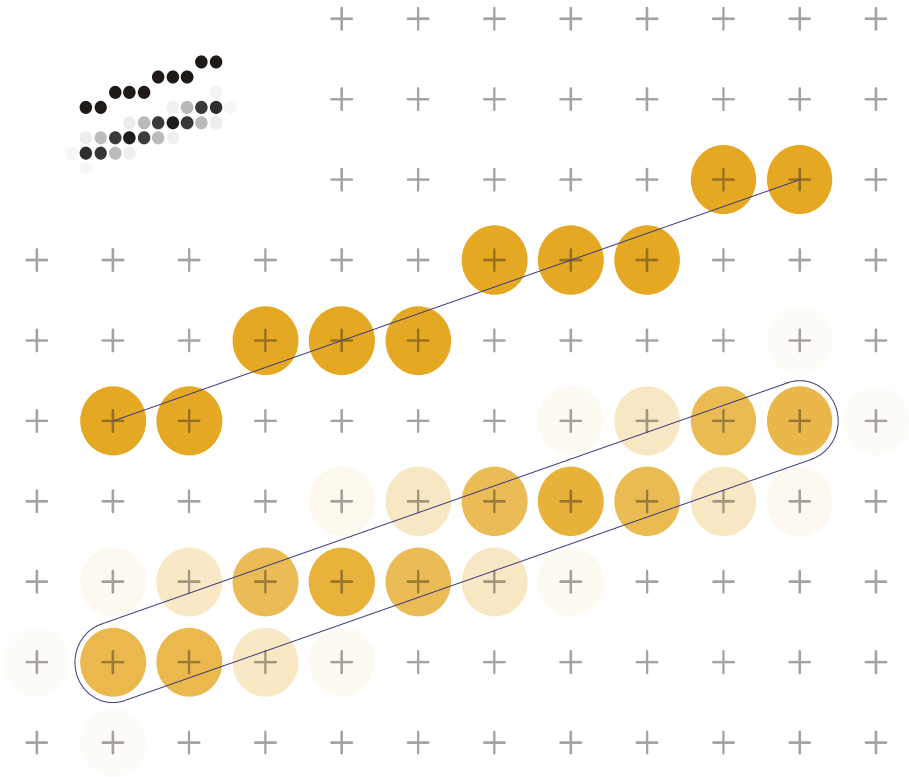
2Δ Σάρωση (Scan Conversion - rasterization)

Andreas Aristidou
andarist@ucy.ac.cy
http://www.andreasaristidou.com

Τι είναι η σάρωση

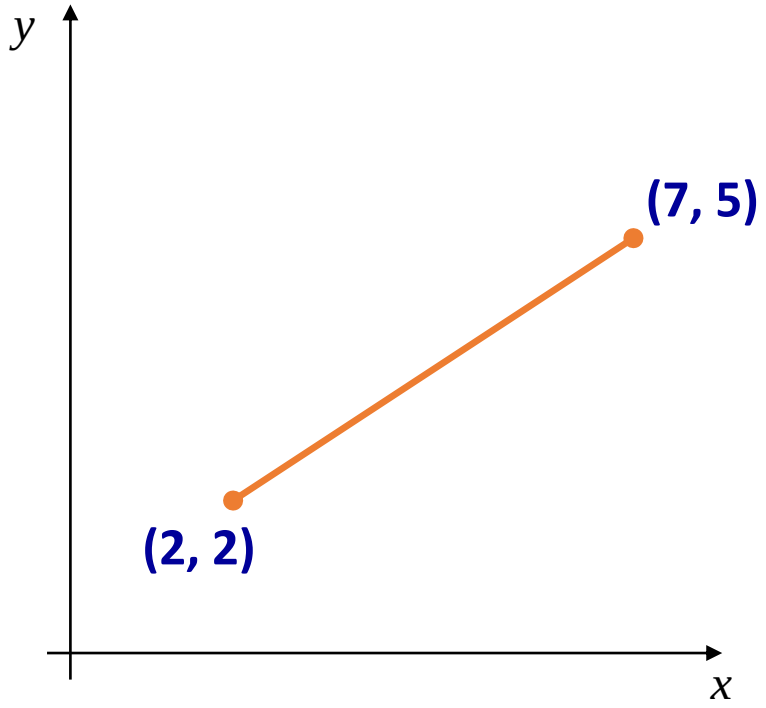
- Είναι το τελευταίο στάδιο του *rasterization* (η διαδικασία κατά την οποία τα γεωμετρικά στοιχεία μετατρέπονται σε πίνακες από pixels και αποθηκεύονται στο framebuffer για να προβληθούν)
- Έπεται της αποκοπής (clipping)
- Όλα τα πακέτα γραφικών κάνουν τη σάρωση στο τέλος του rendering pipeline
- Τρίγωνα (ή υψηλότερης πολυπλοκότητας πολύγωνα) μετατρέπονται σε pixels
- Για 3Δ rendering, λαμβάνουμε υπόψη και άλλες διεργασίες, όπως ο φωτισμός και η σκίαση, αλλά θα επικεντρωθεί πρώτα σε αλγόριθμους για **σάρωση (line scan conversion)**

Αλγόριθμοι σχεδίασης γραμμών



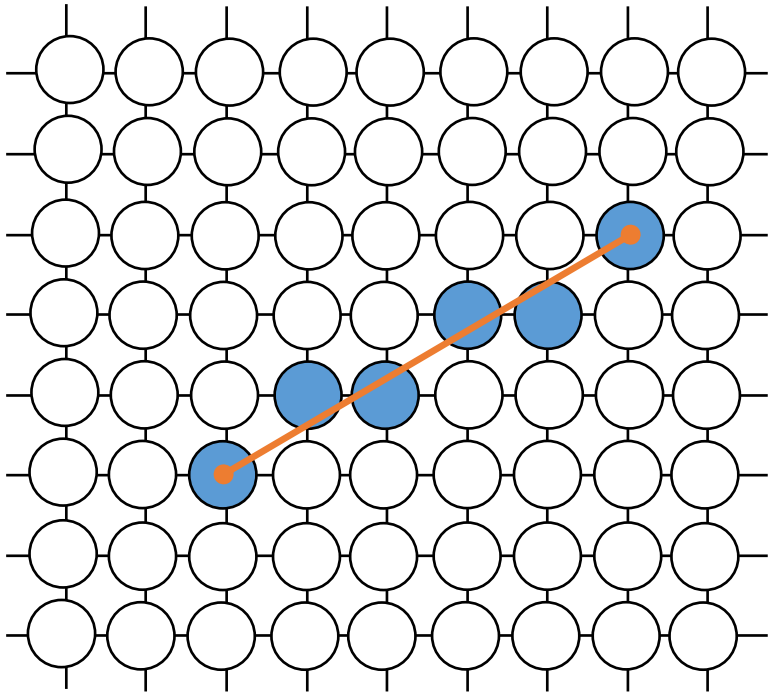
Το πρόβλημα της σάρωσης (scan conversion)

- Ένα τμήμα μιας ευθείας γραμμής σε μια σκηνή ορίζεται από τις συντεταγμένες των θέσεις των τελικών σημείων της γραμμής



Το πρόβλημα της σάρωσης (scan conversion)

- Αλλά τι συμβαίνει όταν προσπαθούμε να αναπαραστήσουμε (rasterize) αυτή τη γραμμή σε μια οθόνη από pixels;



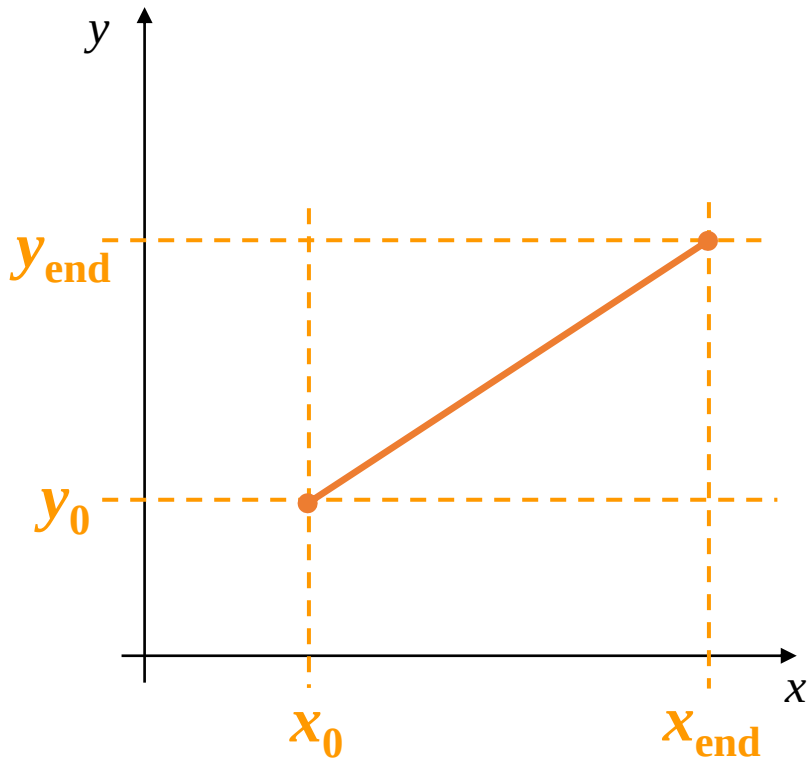
Πώς επιλέγουμε ποια pixels θα ενεργοποιήσουμε;

Προϋποθέσεις

- Προϋποθέσεις που πρέπει να λάβουμε υπόψη μας:
  - Η γραμμή πρέπει να φαίνεται καλή
    - Να μην έχει *jaggies*
  - Πρέπει να είναι πολύ γρήγορος ο αλγόριθμος!
    - Απλά αναλογιστείτε πόσες γραμμές θα έχουμε να ζωγραφίσουμε σε μια σκηνή.

Εξίσωση ευθείας

- Ας εξετάσουμε γρήγορα τις εξισώσεις που χρειάζονται για τη σχεδίαση μιας ευθείας γραμμής



Εξίσωση ευθείας με κλίση

$$y = m \cdot x + b$$

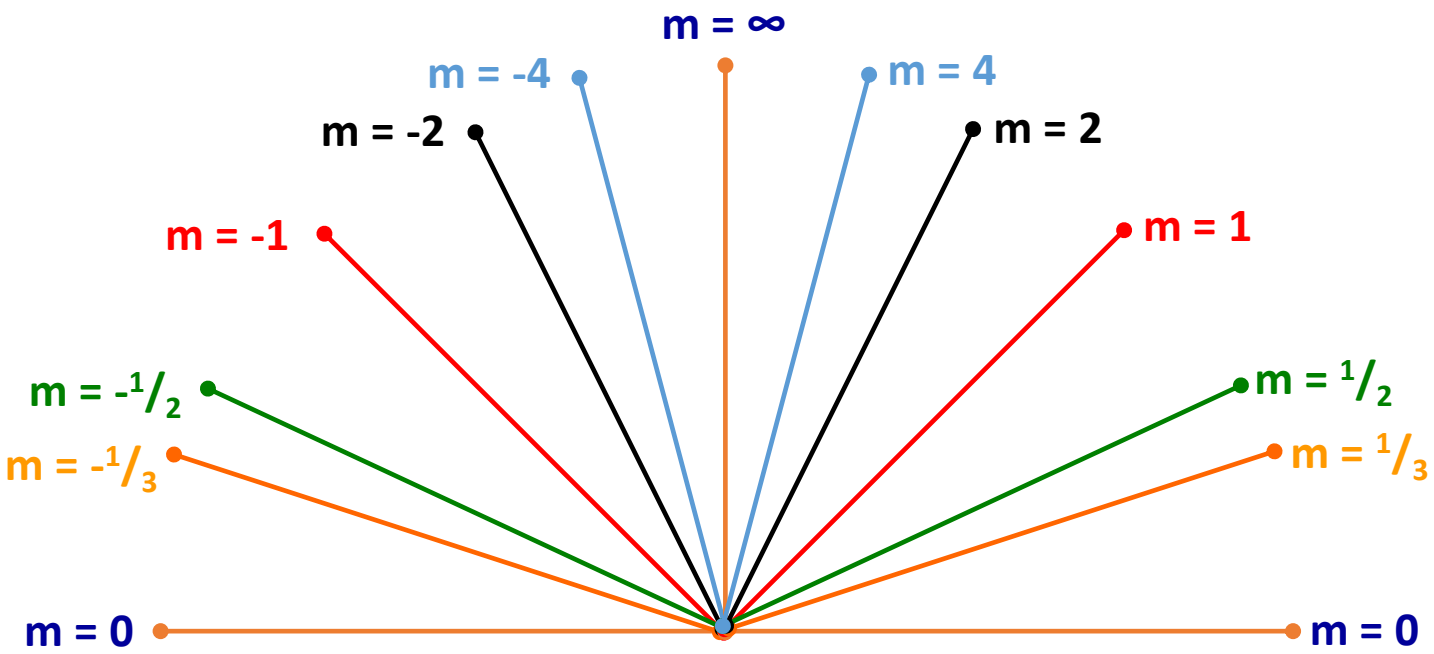
όπου:

$$m = \frac{y_{end} - y_0}{x_{end} - x_0}$$

$$b = y_0 - m \cdot x_0$$

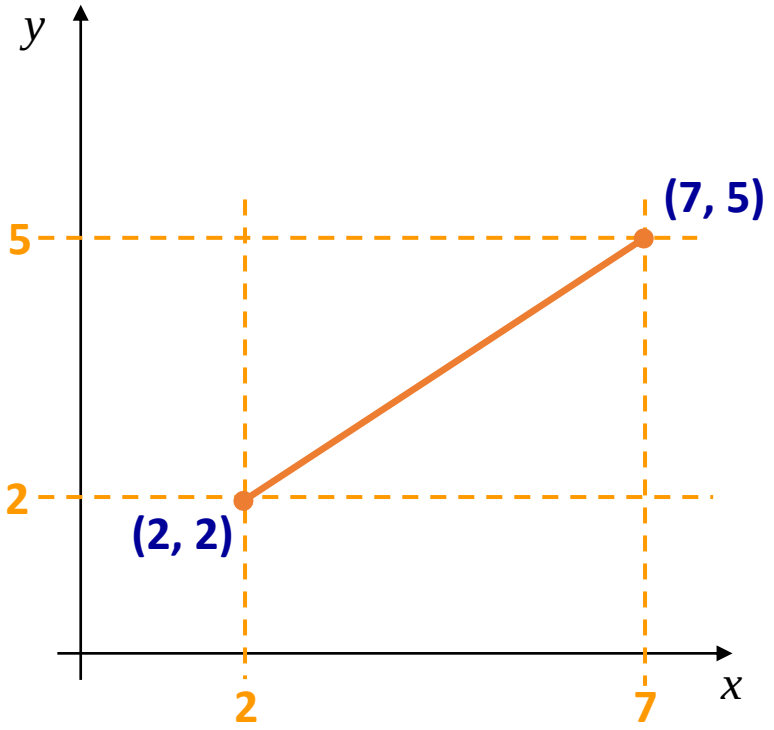
Ευθείες και κλίσεις

- Η κλίση μιας ευθείας (m) ορίζεται από τις συντεταγμένες αρχής και τέλους της ευθείας
- Το παρακάτω διάγραμμα δείχνει μερικά παραδείγματα κλίσεων σε γραμμές



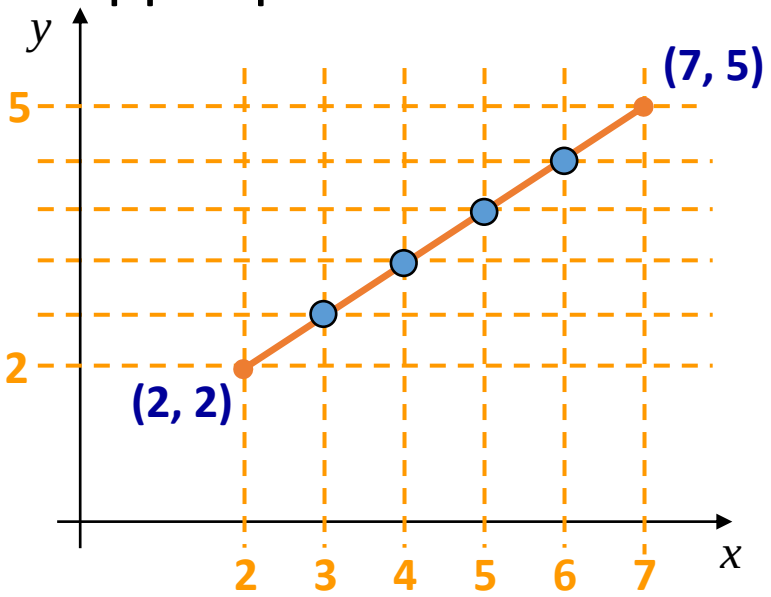
Μια πολύ απλή λύση

- Έστω το πιο κάτω παράδειγμα:



Μια πολύ απλή λύση

- Πρώτα θα βρούμε τα m και b:



$$m = \frac{5-2}{7-2} = \frac{3}{5}$$

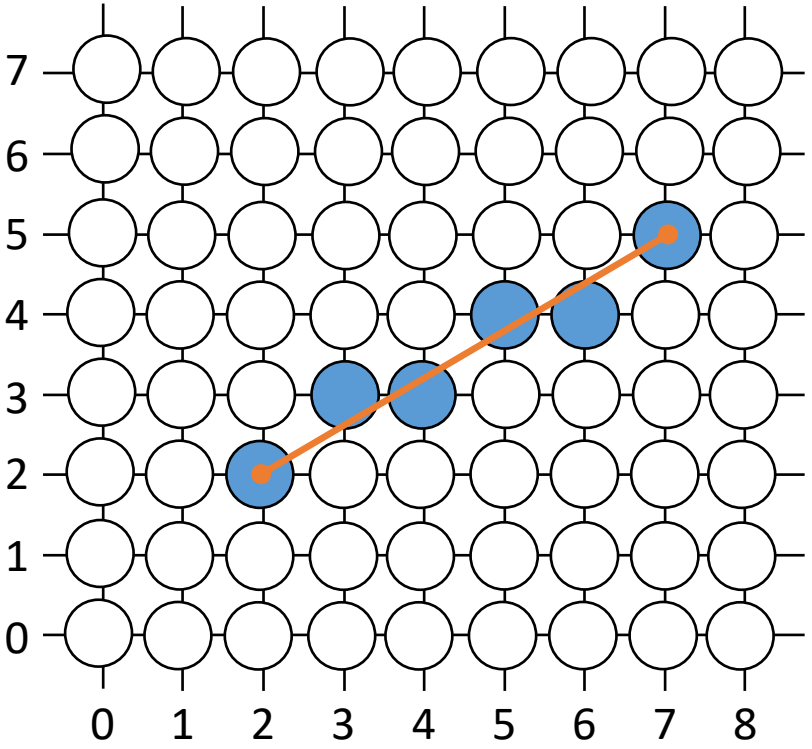
$$b = 2 - \frac{3}{5} * 2 = \frac{4}{5}$$

- Τώρα, για κάθε τιμή του x βρίσκουμε την αντίστοιχη τιμή του y :

$$y(3) = \frac{3}{5} \cdot 3 + \frac{4}{5} = 2\frac{3}{5} \qquad y(4) = \frac{3}{5} \cdot 4 + \frac{4}{5} = 3\frac{1}{5} \qquad y(5) = \frac{3}{5} \cdot 5 + \frac{4}{5} = 3\frac{4}{5} \qquad y(6) = \frac{3}{5} \cdot 6 + \frac{4}{5} = 4\frac{2}{5}$$

Μια πολύ απλή λύση

- Στρογγυλοποιήστε τα αποτελέσματα, και ενεργοποιήστε τα αντίστοιχα pixels



$$y(3) = 2\frac{3}{5} \approx 3$$

$$y(4) = 3\frac{1}{5} \approx 3$$

$$y(5) = 3\frac{4}{5} \approx 4$$

$$y(6) = 4\frac{2}{5} \approx 4$$

Μια πολύ απλή λύση

- Ωστόσο, αυτή η μέθοδος είναι πολύ αργή
- Πιο συγκεκριμένα απαιτείται:
  - Η εξίσωση $y = mx + b$ απαιτεί τον πολλαπλασιασμό του m με το x
  - Στρογγυλοποίηση των αποτελεσμάτων για υπολογισμό του y
- Χρειαζόμαστε μια πιο γρήγορη μέθοδο

Κλίση

- Στο προηγούμενο παράδειγμα επιλέξαμε να λύσουμε την παραμετρική εξίσωση ευθείας για να υπολογίσουμε τις συντεταγμένες του y για κάθε x στοιχείο
- Τι θα γινόταν αν το κάναμε αντίστροφα;
- Τότε θα είχαμε:

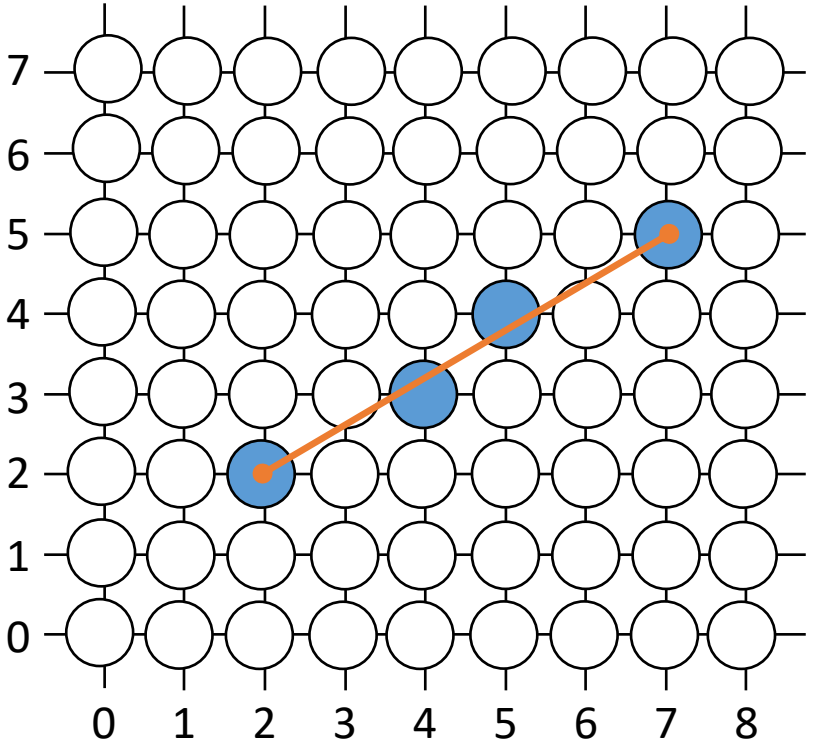
$$x = \frac{y - b}{m}$$

- όπου: $m = \frac{y_{end} - y_0}{x_{end} - x_0}$ και $b = y_0 - m \cdot x_0$

Κλίση

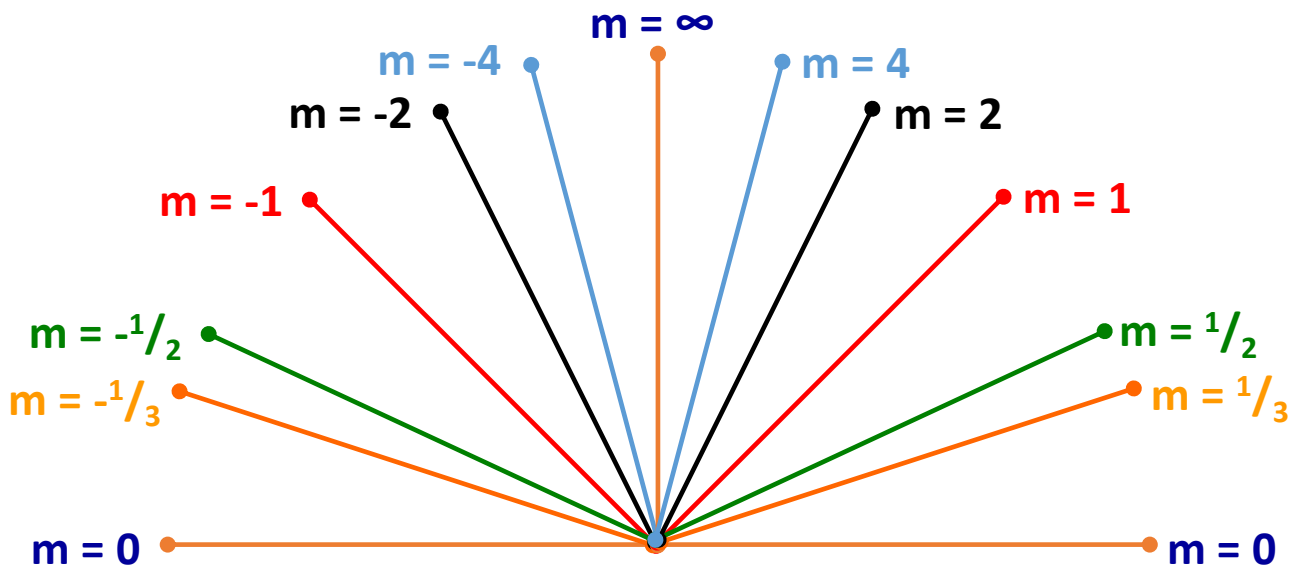
- Αυτό μας δίνει: $x(3) = 3\frac{2}{3} \approx 4$ $x(4) = 5\frac{1}{3} \approx 5$

- Μπορούμε να δούμε εύκολα ότι αυτή η γραμμή δεν φαίνεται πολύ καλή!
- Πως θα μπορούσαμε να απεικονίσουμε την ευθεία σε pixles με χρήση της κλίσης;



Κλίση

- Αν η κλήση είναι μεταξύ του -1 και 1, τότε θα λύσουμε ως προς τις y συντεταγμένες, χρησιμοποιώντας τα x στοιχεία
- Αλλιώς θα κάνουμε το αντίθετο – Οι x συντεταγμένες θα υπολογιστούν βάση των y συντεταγμένων



Ο αλγόριθμος DDA

- Ο *digital differential analyzer* (DDA) αλγόριθμος υιοθετεί μια στοιχειώδη προσέγγιση για να επιταχύνει τη σάρωση
- Απλά υπολόγισε το y_{k+1} βάση του y_k

Ο αλγόριθμος DDA

- Έστω τα πιο κάτω σημείων που καθορίζουν τη γραμμή (από το προηγούμενο παράδειγμά μας):
- $(2, 2), (3, 2^3/_5), (4, 3^1/_5), (5, 3^4/_5), (6, 4^2/_5), (7, 5)$
- Προσέξτε πως, ενώ τα στοιχεία του x αυξάνονται κατά 1, τα στοιχεία του y αυξάνονται όπως και η κλίση
- Αυτό είναι το κύριο συστατικό του αλγόριθμου DDA

Ο αλγόριθμος DDA

- Όταν η κλίση είναι μεταξύ του -1 και 1, τότε ξεκινώντας από το πρώτο στοιχείο της γραμμής, αύξανε το x στοιχείο κατά 1, και υπολόγισε το y στοιχείο ως ακολούθως:

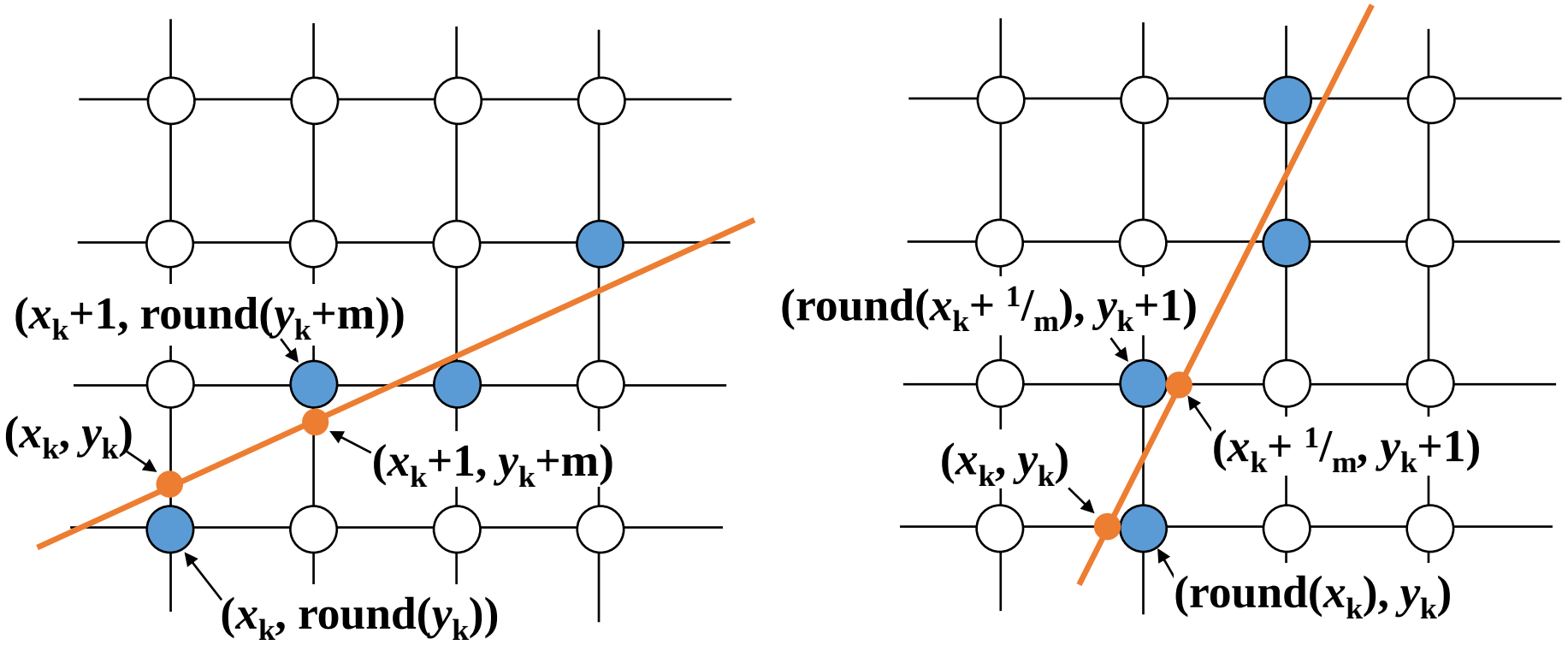
$$y_{k+1} = y_k + m$$

- Όταν η κλίση είναι εκτός αυτών των ορίων, τότε ξεκινώντας από το πρώτο στοιχείο της ευθείας, αύξανε το y στοιχείο κατά 1 και υπολόγισε το αντίστοιχο x στοιχείο ως ακολούθως:

$$x_{k+1} = x_k + \frac{1}{m}$$

Ο αλγόριθμος DDA

- Οι τιμές που υπολογίζονται από τον αλγόριθμο DDA στρογγυλοποιούνται για να ταιριάζουν στις θέσεις του pixel



Ο αλγόριθμος DDA

- Ο αλγόριθμος DDA είναι πολύ πιο γρήγορος από τις προηγούμενες λύσεις που έχουμε δει
  - Ειδικότερα, δεν εμπλέκονται πλέον πολλαπλασιασμοί
- Ωστόσο, εξακολουθούν να υπάρχουν δύο μεγάλα ζητήματα:

```
void Line(int x0, int y0, int x1, int y1) {
    int   x, y;
    float dy = y1 - y0;
    float dx = x1 - x0;
    float m  = dy / dx;

    y = y0;
    for (x = x0; x < x1; ++x) {
        WritePixel( x, Round(y) );
        y = y + m;
    }
}
```

Since slope is fractional, need special case for vertical lines (dx = 0)

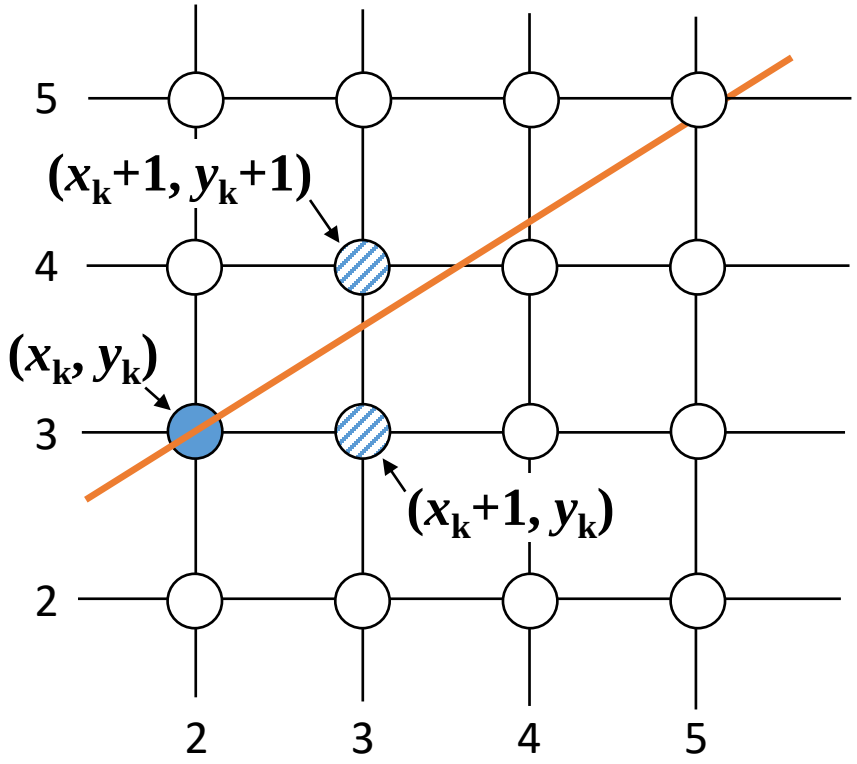
Rounding takes time

Ο αλγόριθμος ευθείας Bresenham

- Ο αλγόριθμος Bresenham είναι ένας ακόμη αλγόριθμος σάρωσης
- Το μεγάλο πλεονέκτημα αυτού του αλγορίθμου είναι ότι χρησιμοποιεί μόνο ακέραιους υπολογισμούς

Η βασική ιδέα

- Μετακινηθείτε κατά μήκος του x άξονα σε μοναδιαία διαστήματα και σε κάθε περίπτωση επέλεξε μεταξύ των δύο y συντεταγμένων

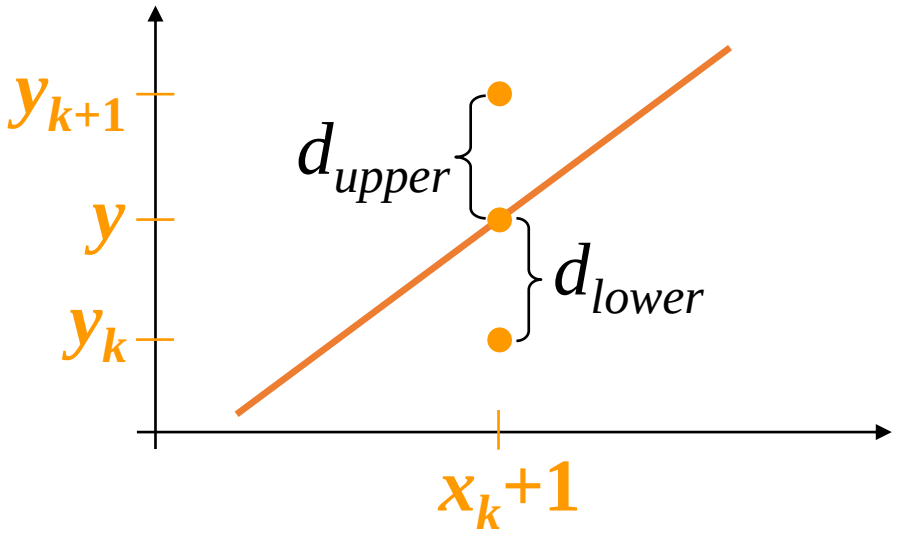


Για παράδειγμα, από τη θέση (2, 3) έχουμε να επιλέξουμε μεταξύ του (3, 3) και (3, 4)

Θέλουμε το σημείο που είναι πιο κοντά στην ευθεία μας

Ο αλγόριθμος ευθείας Bresenham

- Στο σημείο x_k+1, ο κάθετος διαχωρισμός ορίζεται σαν d_{upper} και d_{lower}



Το y στοιχείο στο σημείο x_k+1 είναι: $y = m(x_k + 1) + b$

Ο αλγόριθμος ευθείας Bresenham

- Οπότε, τα d_{upper} και d_{lower} είναι:

$$d_{lower} = y - y_k$$
$$= m(x_k + 1) + b - y_k$$

- και:

$$d_{upper} = (y_k + 1) - y$$
$$= y_k + 1 - m(x_k + 1) - b$$

- Χρησιμοποιώντας αυτό μπορούμε εύκολα να αποφασίσουμε ποιο pixel είναι πιο κοντά στην ευθεία

Ο αλγόριθμος ευθείας Bresenham

- Αυτή η απόφαση μπορεί πιο εύκολα να υπολογιστεί ως εξής:

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1$$

- Αντικαθιστώντας το m με το $\Delta y/\Delta x$ όπου Δx και Δy είναι οι διαφορές μεταξύ των τελικών σημείων:

$$\Delta x(d_{lower} - d_{upper}) = \Delta x(2\frac{\Delta y}{\Delta x}(x_k + 1) - 2y_k + 2b - 1)$$

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + 2\Delta y + \Delta x(2b - 1)$$

$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

Ο αλγόριθμος ευθείας Bresenham

- Οπότε η απόφαση p_k για το kth βήμα είναι:

$$p_k = \Delta x(d_{lower} - d_{upper})$$
$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c$$

- Το πρόσημο της παραμέτρου απόφασης p_k είναι το ίδιο με την πράξη $d_{lower} - d_{upper}$
- Εάν το p_k είναι αρνητικό, τότε επιλέγουμε το κάτω pixel, αλλιώς επιλέγουμε το πάνω

Ο αλγόριθμος ευθείας Bresenham

- Θυμηθείτε πως τα x στοιχεία αυξάνονται μοναδιαία
- Οπότε στο βήμα k+1 η παράμετρος απόφασης είναι:

$$p_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

- Αντικαθιστώντας το p_k θα έχουμε:

$$p_{k+1} - p_k = 2\Delta y(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

Ο αλγόριθμος ευθείας Bresenham

- Όμως το, x_{k+1} είναι ίσο με $x_k + 1$, οπότε:

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k)$$

- όπου y_{k+1} - y_k είναι είτε 0 ή 1, εξαρτάται από το πρόσημο του p_k
- Η πρώτη παράμετρος απόφασης p0 (στο x0, y0) δίνεται από την εξίσωση:

$$p_0 = 2\Delta y - \Delta x$$

Ο αλγόριθμος ευθείας Bresenham

BRESENHAM'S LINE DRAWING ALGORITHM
(για $|m| < 1.0$)

1. Εισάγετε τα δύο σημεία τέλους της γραμμής, αποθηκεύοντας το αριστερό σημείο ως (x_0, y_0)
2. Σχεδιάστε το σημείο (x_0, y_0)
3. Υπολογίστε τις σταθερές Δx, Δy, $2\Delta y$, και $(2\Delta y - 2\Delta x)$ και υπολογίστε την πρώτη τιμή για την παράμετρο απόφασης ως:
$$p_0 = 2\Delta y - \Delta x$$
4. Για κάθε x_k, ξεκινώντας με $k = 0$, υπολογίστε τα ακόλουθα. Εαν $p_k < 0$, το επόμενο σημείο είναι το (x_k+1, y_k) και:
$$p_{k+1} = p_k + 2\Delta y$$

Ο αλγόριθμος ευθείας Bresenham

- **Σημείωση!** Ο αλγόριθμος προϋποθέτει ότι οι κλίση είναι μικρότερη του 1. Για διαφορετικές κλίσεις πρέπει να προσαρμόσουμε ελαφρώς τον αλγόριθμο

Αλλιώς, το επόμενο σημείο θα είναι το (x_k+1, y_k+1) και:

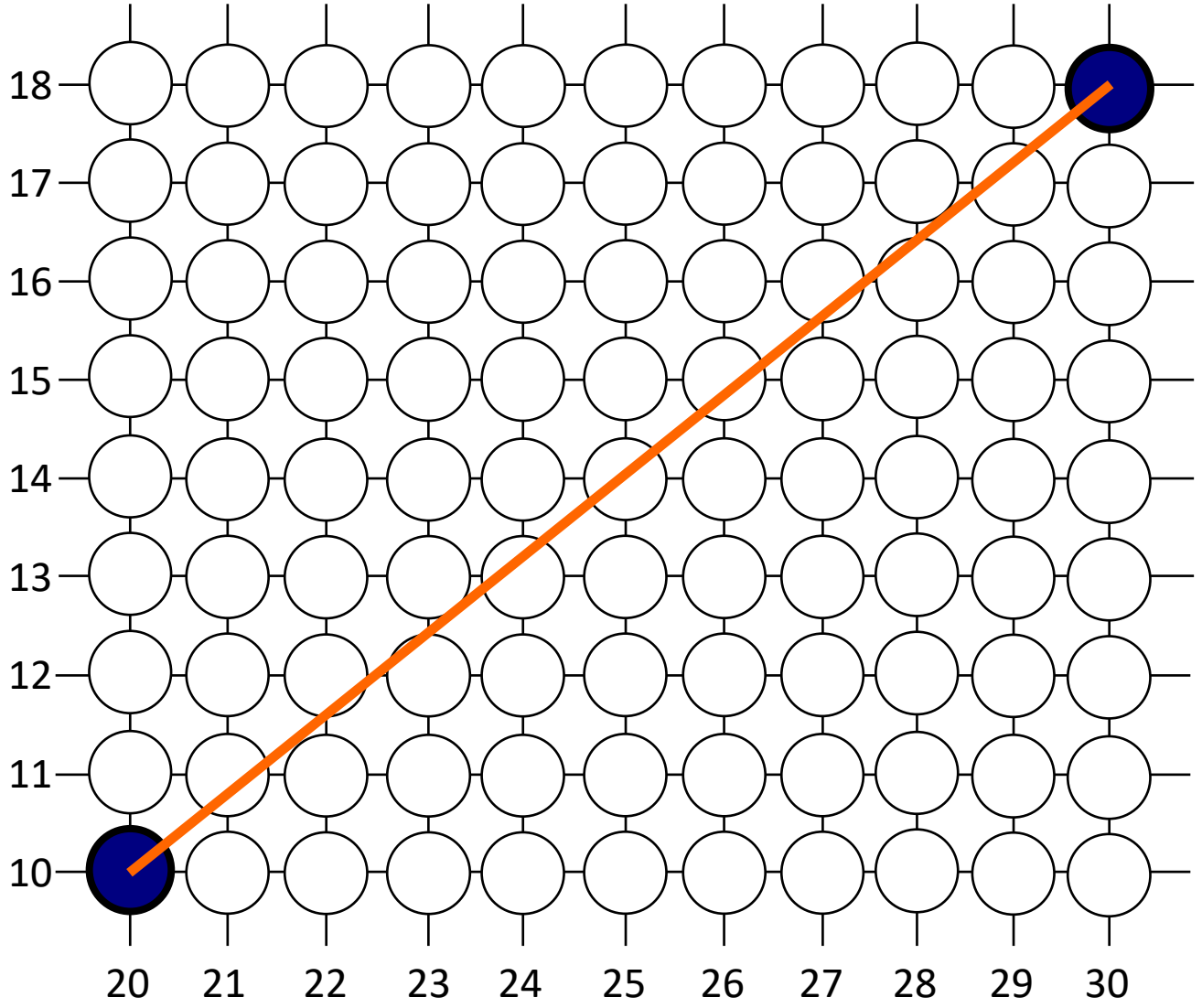
$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$

5. Επαναλάβετε το βήμα 4 για $(\Delta x - 1)$ φορές

Παράδειγμα υλοποίησης του αλγόριθμου Bresenham

- Ας προβάλουμε την ευθεία από το σημείο (20, 10) στο (30, 18)
- Πρώτα υπολογίζουμε όλες τις σταθερές:
  - Δx: 10
  - Δy: 8
  - $2\Delta y$: 16
  - $2\Delta y - 2\Delta x$: -4
- και την αρχική παράμετρο απόφασης p_0:
  - $p0 = 2\Delta y - \Delta x$ = 6

Παράδειγμα υλοποίησης του αλγόριθμου Bresenham

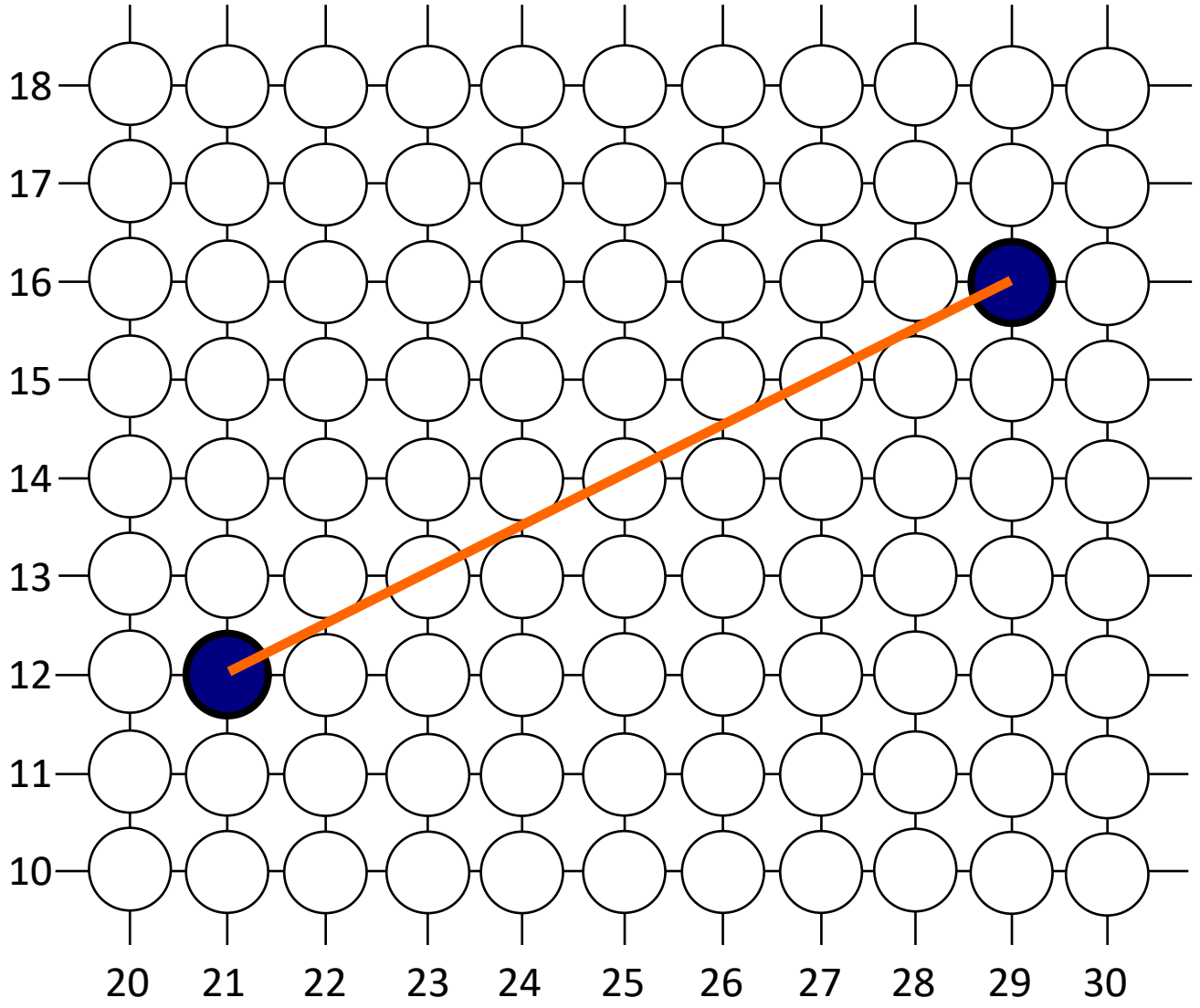


k	p_k	(x_{k+1},y_{k+1})
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

Παράδειγμα υλοποίησης του αλγόριθμου Bresenham

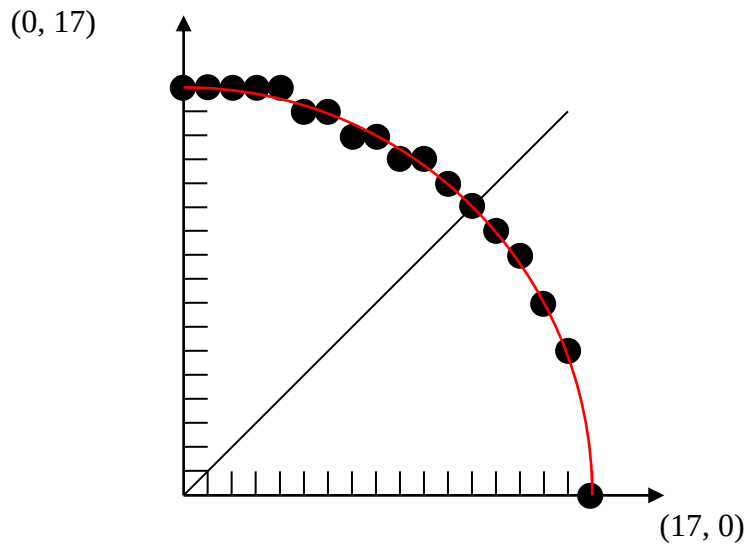
- Χρησιμοποιείστε τον αλγόριθμο Bresenham για την ευθεία που ξεκινά και τελειώνει στα σημεία (21,12) και (29,16) αντίστοιχα

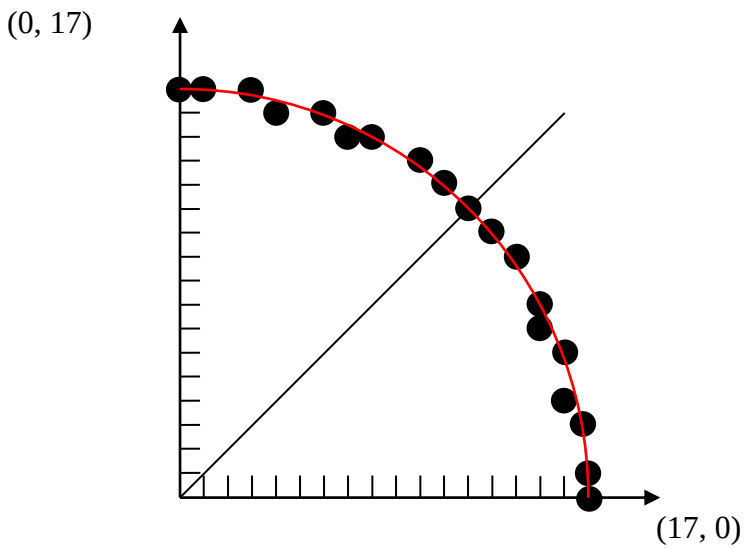
Παράδειγμα υλοποίησης του αλγόριθμου Bresenham



k	p_k	(x_{k+1},y_{k+1})
0		
1		
2		
3		
4		
5		
6		
7		
8		

Αλγόριθμοι σχεδίασης κύκλων





Ένας απλός τρόπος σχεδίασης κύκλων

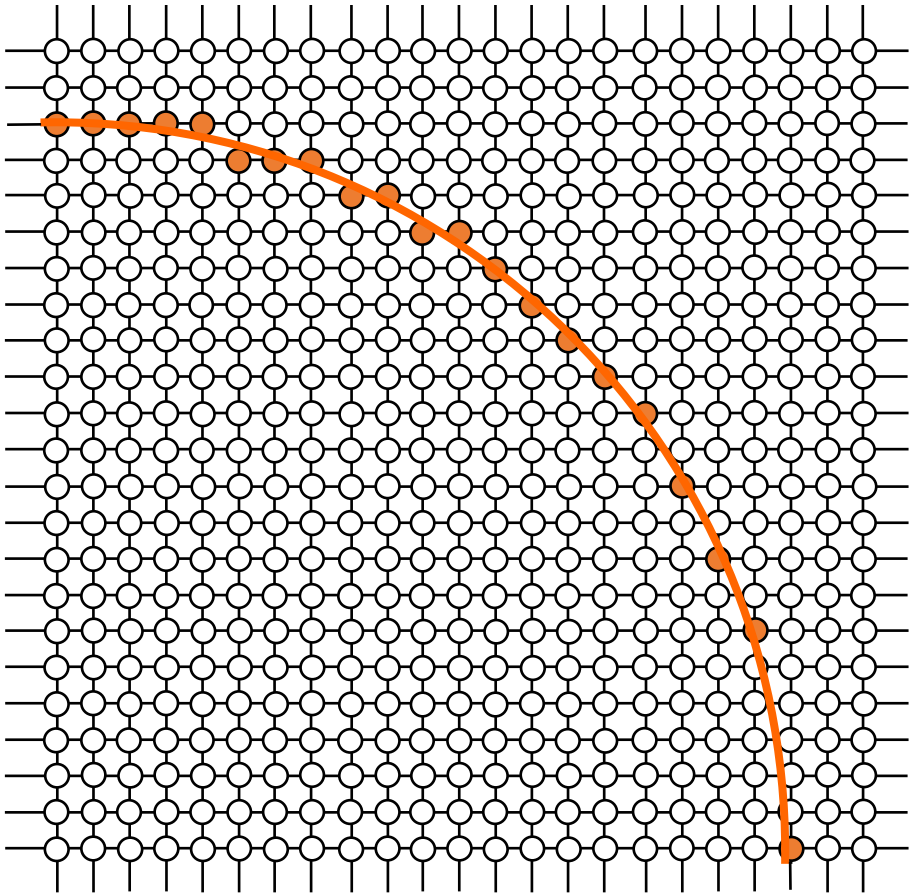
- Η εξίσωση κύκλου είναι:

$$x^2 + y^2 = r^2$$

- όπου r είναι η ακτίνα του κύκλου
- Οπότε, μπορούμε να λύσουμε το y για κάθε μοναδιαίο βήμα του x :

$$y = \pm\sqrt{r^2 - x^2}$$

Ένας απλός τρόπος σχεδίασης κύκλων



$$y_0 = \sqrt{20^2 - 0^2} \approx 20$$

$$y_1 = \sqrt{20^2 - 1^2} \approx 20$$

$$y_2 = \sqrt{20^2 - 2^2} \approx 20$$

$$\vdots$$

$$y_{19} = \sqrt{20^2 - 19^2} \approx 6$$

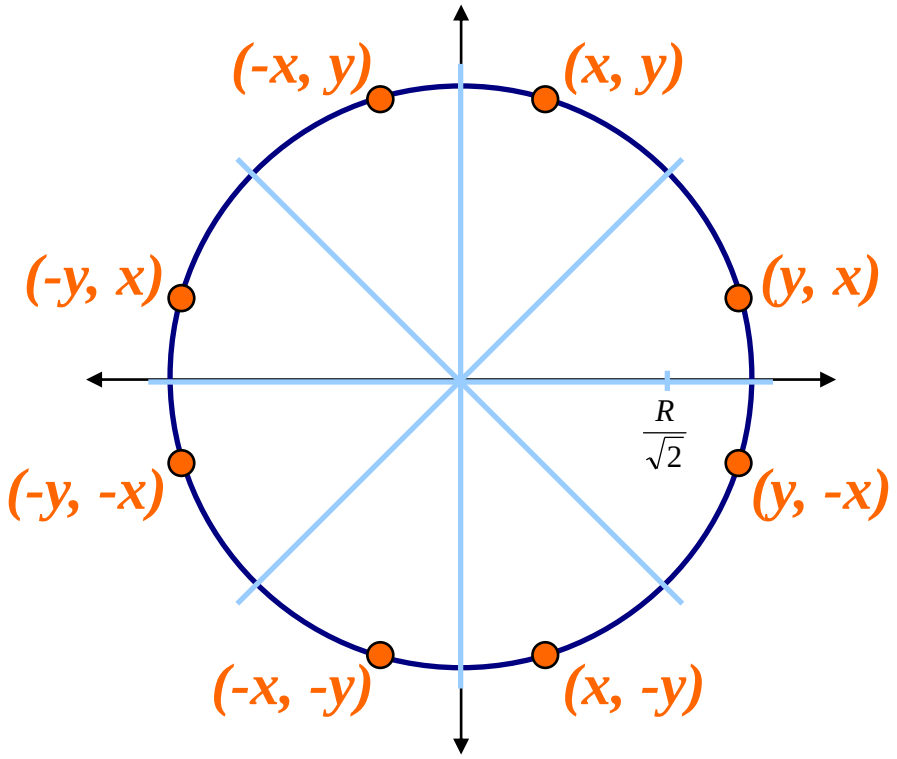
$$y_{20} = \sqrt{20^2 - 20^2} \approx 0$$

Ένας απλός τρόπος σχεδίασης κύκλων

- Ωστόσο, όπως αναμενόταν, αυτό δεν είναι μια καλή λύση!
- Πρώτον, ο κύκλος που προκύπτει έχει μεγάλα κενά όταν η κλίση γίνεται κάθετη
- Δεύτερον, οι υπολογισμοί δεν είναι πολύ αποτελεσματικοί
  - Υπολογισμός τετραγώνου
  - Υπολογισμός της ρίζας – Προσπαθήστε γενικά να αποφεύγεται τέτοιες πράξεις!
- Χρειαζόμαστε μια πιο αποτελεσματική, και ακριβέστερη λύση

Οκταπλή συμμετρία

- Το πρώτο πράγμα που μπορούμε να παρατηρήσουμε σε ένα κύκλο με επίκεντρο το $(0, 0)$ είναι ότι έχουμε οκταπλή συμμετρία

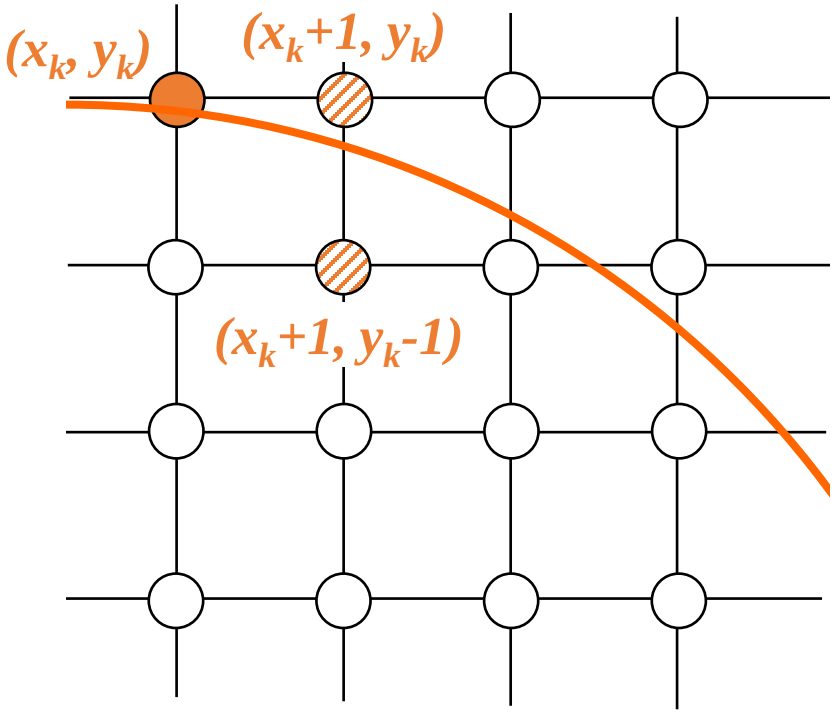


Ο αλγόριθμος κύκλου Mid-Point

- Ομοίως με την περίπτωση των γραμμών, υπάρχει ένας σειριακός αλγόριθμος για το σχεδιασμό κύκλου – ο ***mid-point circle algorithm***
- Σε αυτό τον αλγόριθμο χρησιμοποιούμε την οκταπλή συμμετρία για να υπολογίσουμε τα πάνω οκτώ σημεία του κύκλου, ενώ τα υπόλοιπα υπολογίζονται απλά χρησιμοποιώντας τη συμμετρία

Mid-Point Circle Algorithm

- Έστω ότι τυπώσαμε το σημείο (x_k, y_k)
- Το επόμενο σημείο είναι μεταξύ των (x_k+1, y_k) και (x_k+1, y_k-1)
- Πρέπει να επιλέξουμε αυτό που είναι πιο κοντά στον κύκλο
- Πως παίρνουμε αυτή την απόφαση;



Mid-Point Circle Algorithm

- Ας τροποποιήσουμε την εξίσωση του κύκλου για να μας δώσει:

$$f_{circ}(x,y) = x^2 + y^2 - r^2$$

- Η εξίσωση αξιολογείται ως εξής:

$$f_{circ}(x,y)\begin{cases} < 0, & \text{εάν το } (x,y) \text{ είναι μέσα στον κύκλο} \\ = 0, & \text{εάν το } (x,y) \text{ είναι πάνω στον κύκλο} \\ > 0, & \text{εάν το } (x,y) \text{ είναι έξω από τον κύκλο} \end{cases}$$

- Με την αξιολόγηση αυτής της λειτουργίας στο ενδιάμεσο σημείο μεταξύ των υποψηφίων pixels μπορούμε να πάρουμε την απόφασή μας

Mid-Point Circle Algorithm

- Έστω έχουμε τα σημεία (x_k, y_k) και πρέπει να επιλέξουμε μεταξύ των (x_k+1, y_k) και (x_k+1, y_k-1)
- Η απόφαση μας καθορίζεται από το : $p_k = f_{circ}(x_k+1, y_k - \frac{1}{2})$

$$= (x_k+1)^2 + (y_k - \frac{1}{2})^2 - r^2$$

- Εάν το $p_k < 0$ το μέσο σημείο είναι μέσα στον κύκλο και το y_k είναι πιο κοντά στον κύκλο
- Αλλιώς το σημείο είναι εκτός του κύκλου, οπότε το y_k-1 είναι πιο κοντά

Mid-Point Circle Algorithm

- Για να διασφαλίσουμε ότι τα πράγματα είναι όσο το δυνατόν πιο αποτελεσματικά μπορούμε να κάνουμε όλους τους υπολογισμούς μας βαθμιαία
- Οπότε:

$$p_{k+1} = f_{circ}\left(x_{k+1}+1, y_{k+1}-\tfrac{1}{2}\right)$$

$$= [(x_k+1)+1]^2 + \left(y_{k+1}-\tfrac{1}{2}\right)^2 - r^2$$

- ή: $$p_{k+1} = p_k + 2(x_k+1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$

- όπου y_{k+1} είναι είτε y_k ή y_k-1 ανάλογα με το πρόσημο του p_k

Mid-Point Circle Algorithm

- Η πρώτη παράμετρος απόφασης δίνεται ως: $p_0 = f_{circ}(1, r - 1/2)$

$$= 1 + (r - 1/2)^2 - r^2$$

$$= 5/4 - r$$

- Ακολούθως, αν $p_k < 0$ η επόμενη παράμετρος απόφασης δίνεται ως:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

- Εαν $p_k > 0$ τότε:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_k + 1$$

Mid-Point Circle Algorithm

MID-POINT CIRCLE ALGORITHM

1. Έχοντας την ακτίνα r και το κέτρο του κύκλου (x_c, y_c), μπορούμε εύκολα να υπολογίσουμε τις συντεταγμένες του πρώτου σημείου:

$$(x_0, y_0) = (0, r)$$

2. Η αρχική τιμή της παραμέτρου απόφασης :

$$p_0 = 5/4 - r$$

3. Ξεκινώντας για $k = 0$ για κάθε x_k, εφαρμόζουμε τα ακόλουθα. Εάν $p_k < 0$, το επόμενο σημείο στον κύκλο είναι το (x_k+1, y_k) και:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

The Mid-Point Circle Algorithm

- αλλιώς το σημείο του κύκλου θα είναι το (x_k+1, y_k-1) και:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

4. Προσδιορίστε τα σημεία συμμετρίας στα άλλα επτά τεμάχια
5. Για κάθε στοιχείο (x, y) στον κύκλο με κέντρο (x_c, y_c) υπολόγισε τα σημεία του:

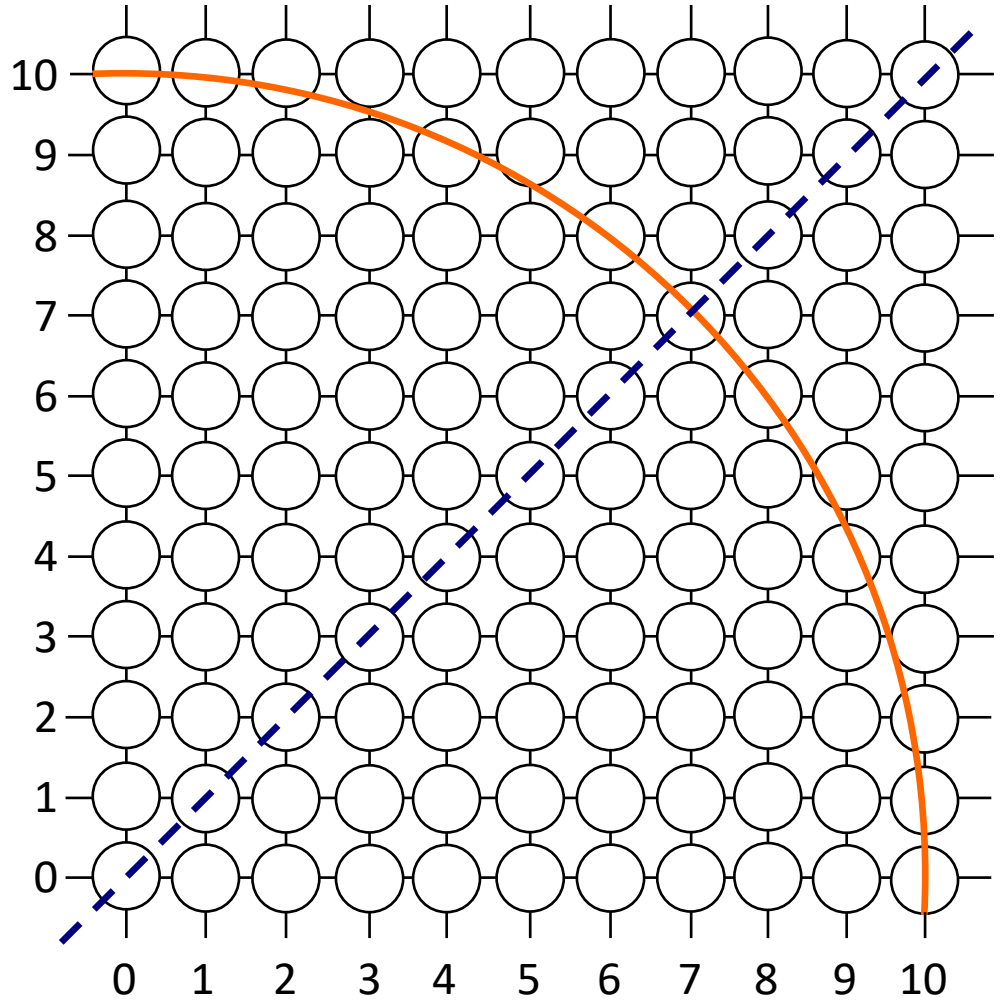
$$x = x + x_c \qquad y = y + y_c$$

6. Επανέλαβε τα βήματα 3 έως 5 μέχρι $x >= y$

Παράδειγμα του Mid-Point Circle Algorithm

- Έστω ο κύκλος με κέντρο το (0,0) και ακτίνα 10

Παράδειγμα του Mid-Point Circle Algorithm

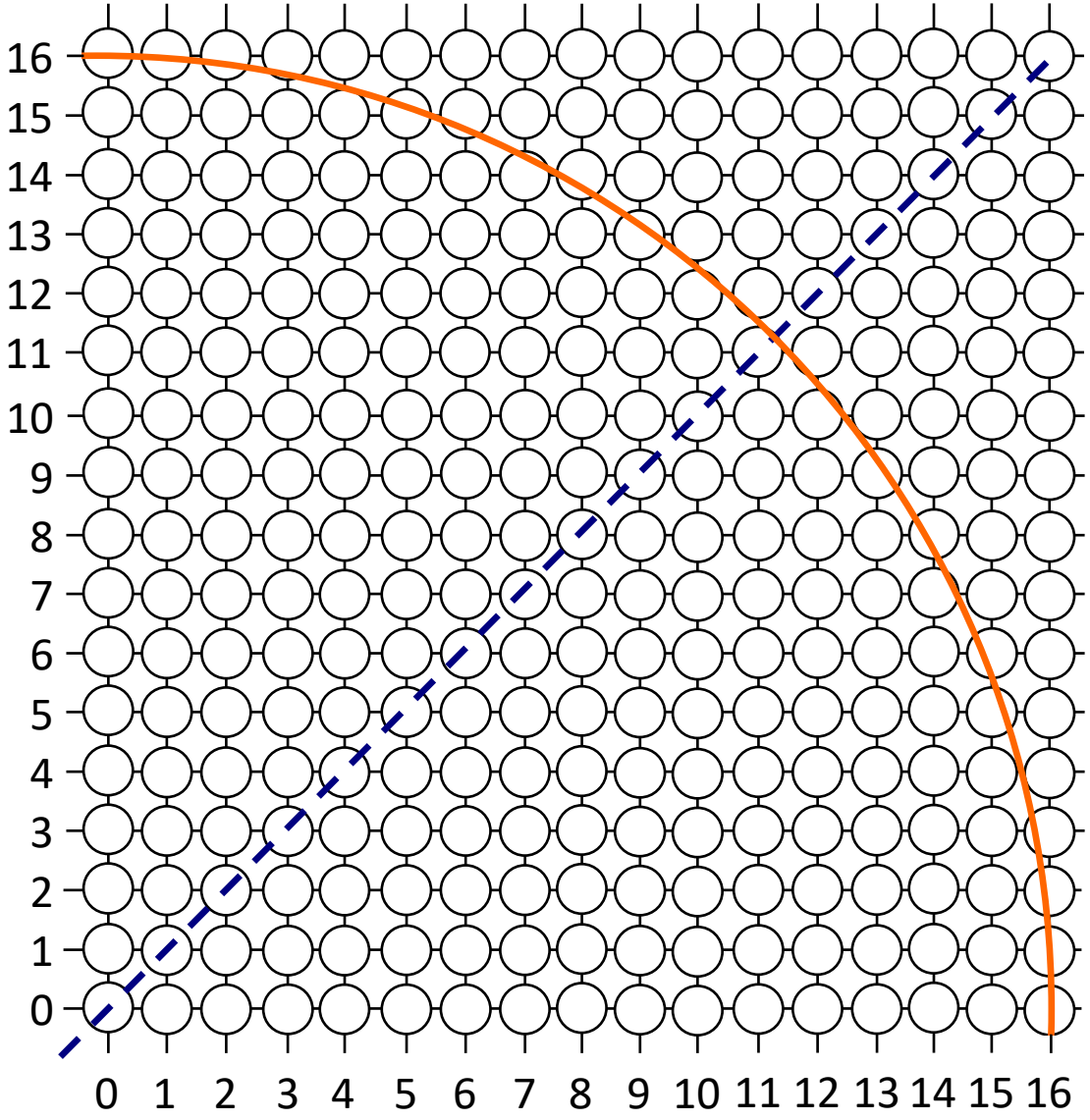


k	p_k	(x_{k+1},y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0				
1				
2				
3				
4				
5				
6				

Παράδειγμα του Mid-Point Circle Algorithm

- Χρησιμοποιείστε τον mid-point circle algorithm για να τυπώσετε τον κύκλο με κέντρο (0,0) και ακτίνα 15

Παράδειγμα του Mid-Point Circle Algorithm



k	p_k	(x_{k+1},y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0				
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				

Σύνοψη του Mid-Point Circle Algorithm

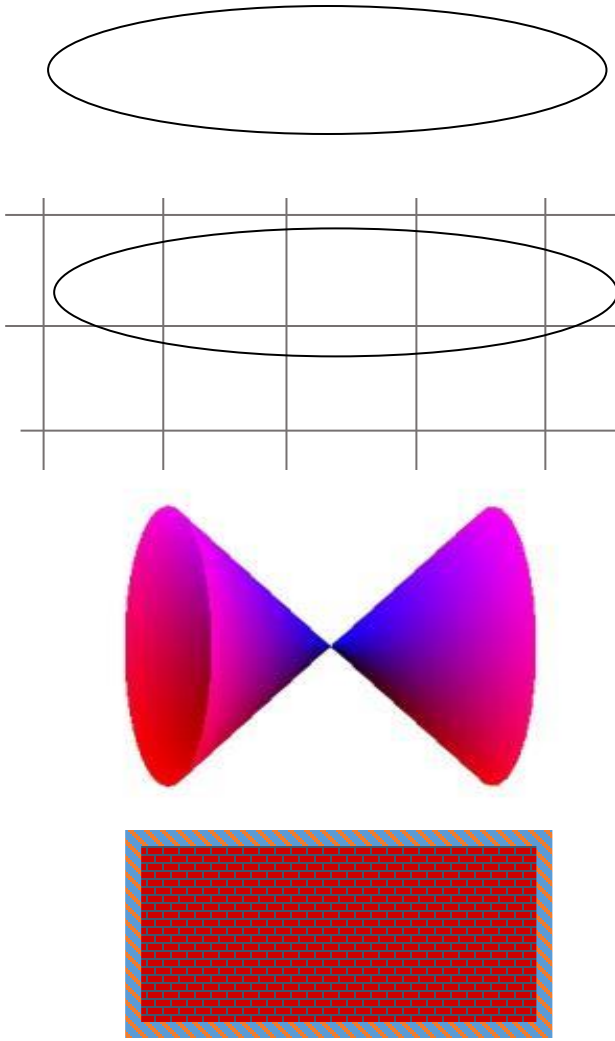
- Το βασικό κλειδί στον αλγόριθμο του κύκλου είναι:
  - Η οκταπλή συμμετρία μειώνει τον χρόνο που απαιτείται για τον υπολογισμό
  - Κινούμαστε κατά μήκος του x άξονα για τον υπολογισμό του κύκλου επιλέγοντας μεταξύ δύο πιθανών σημείων για την συντεταγμένη του y

Midpoint Eighth Circle Algorithm

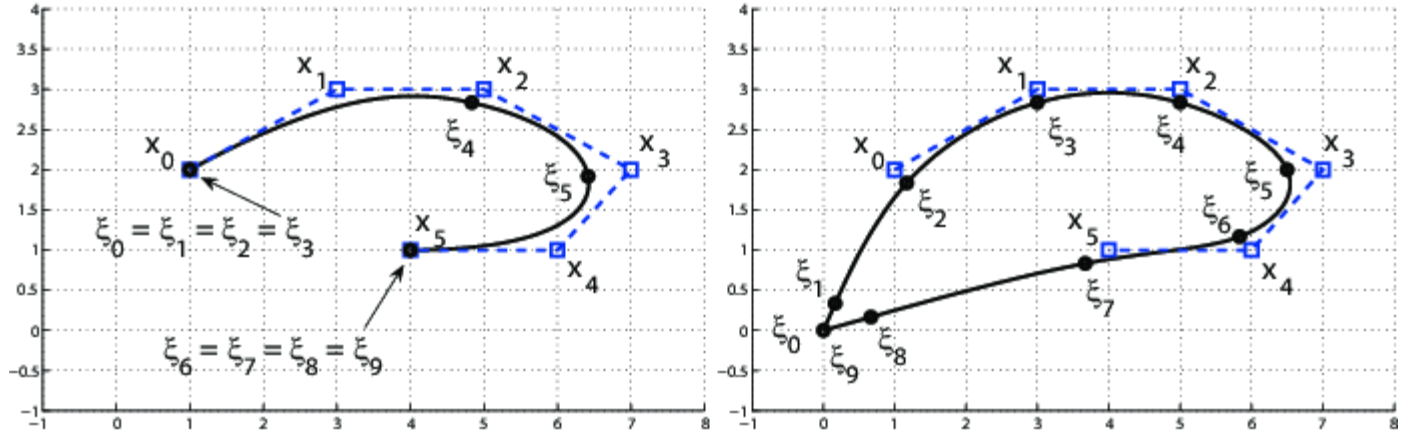
```
MidpointEighthCircle(R) { /* 1/8th of a circle w/ radius R */
    int x = 0, y = R;
    int deltaE   = 2 * x + 3;
    int deltaSE  = 2 * (x - y) + 5;
    float decision = (x + 1) * (x + 1) + (y - 0.5) * (y - 0.5) - R*R;
    WritePixel(x, y);

    while ( y > x ) {
        if (decision > 0) { // Move East
            x++; WritePixel(x, y);
            decision += deltaE;
            deltaE += 2; deltaSE += 2; // Update deltas
        } else { // Move SouthEast
            y--; x++; WritePixel(x, y);
            decision += deltaSE;
            deltaE += 2; deltaSE += 4; // Update deltas
        }
    }
}
```

Άλλα Scan-conversion προβλήματα

- Aligned Ellipses
- Non-integer primitives
- General conics
- Patterned primitives

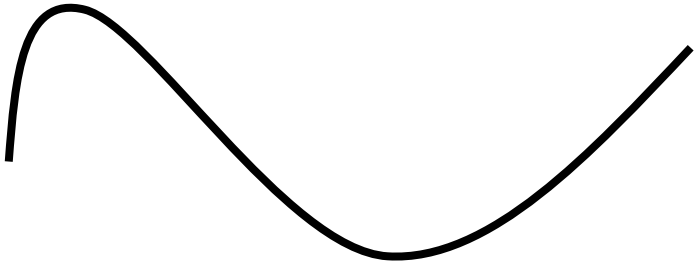


Spline Representations

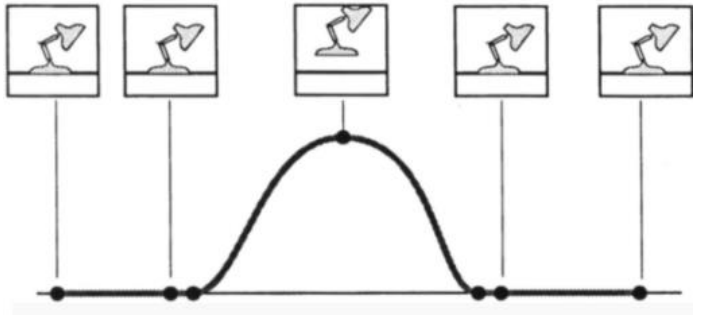


Spline Representations

- Μια ομαλή καμπύλη (spline) ορίζεται μαθηματικά χρησιμοποιώντας ένα σύνολο περιορισμών
- Οι καμπύλες έχουν πολλές χρήσεις:
  - 2D illustration
  - Fonts
  - 3D Modelling
  - Animation



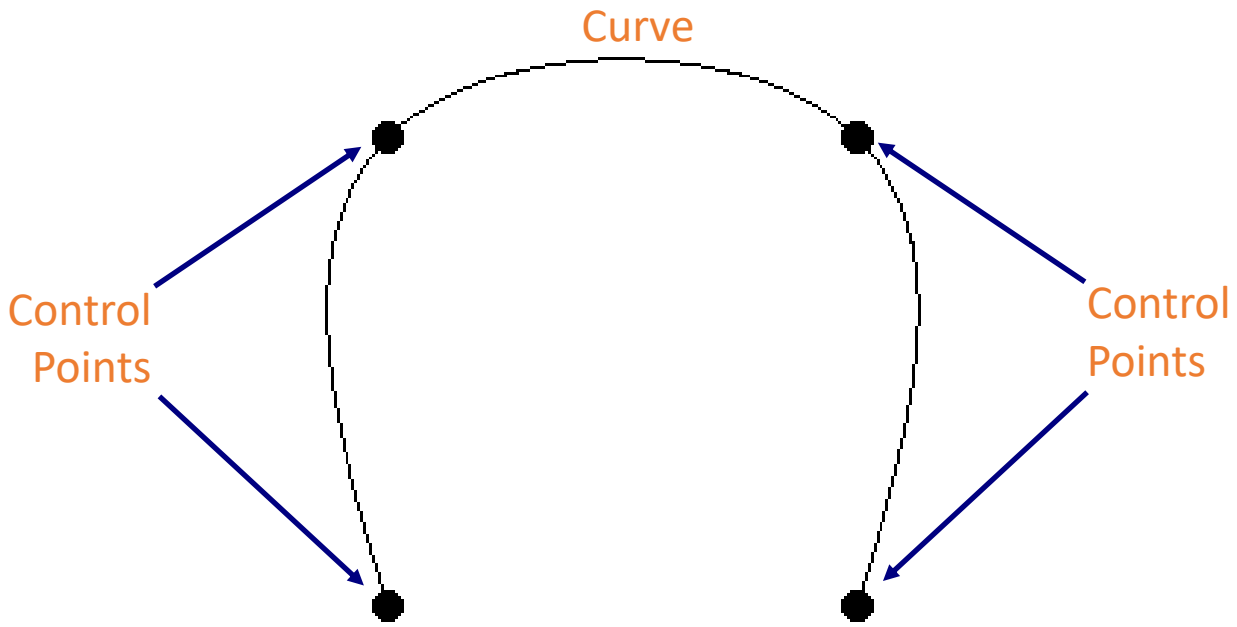
"Manifold Splines", X. Gu, Y. He & H. Qin, Solid and Physics Modeling 2005.



ACM © 1987 "Principles of traditional animation applied to 3D computer animation"

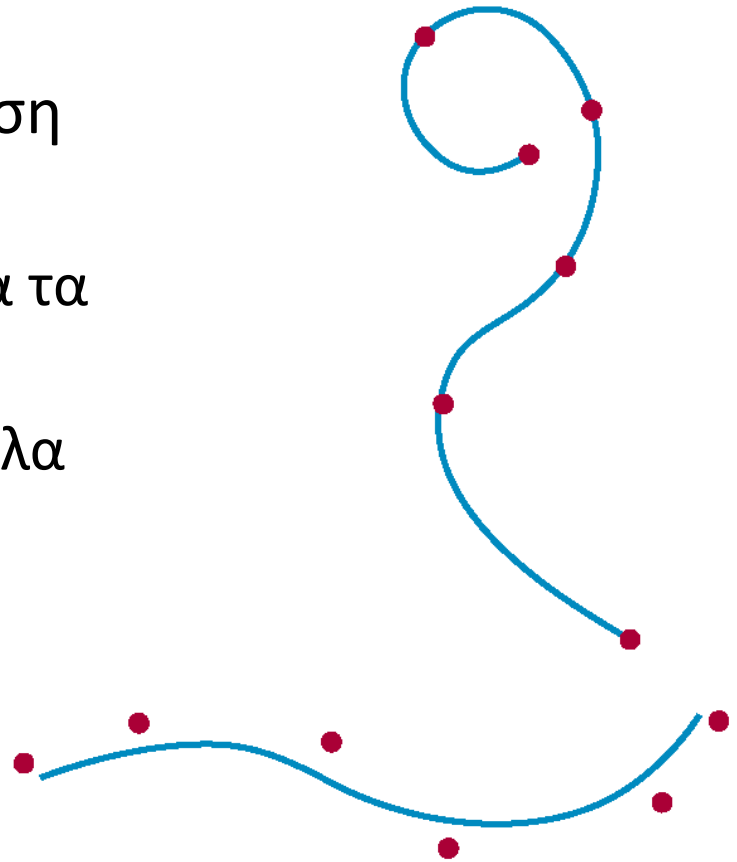
Η βασική ιδέα

- Ο χρήστης καθορίζει τα σημεία ελέγχου
- Ορίζεται μια ομαλή καμπύλη



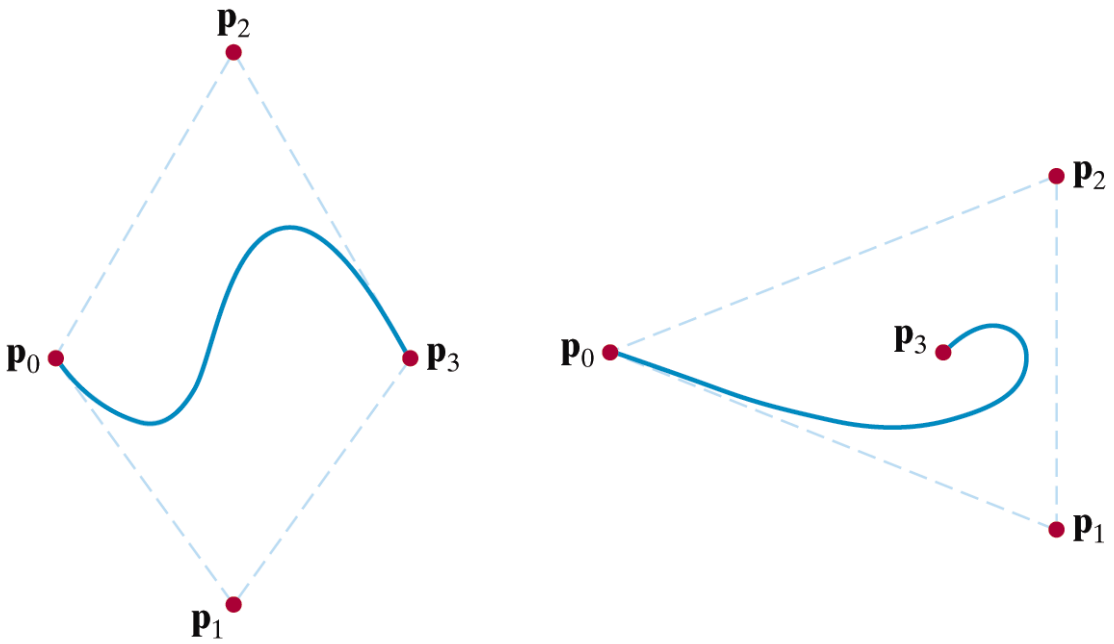
Interpolation Vs Approximation

- Η καμπύλη ορίζεται από ένα σετ από **σημεία ελέγχου**
- Υπάρχουν 2 τρόποι να ορισθεί η καμπύλη βάση αυτών των σημείων
  - **Interpolation -** η καμπύλη περνά μέσα από όλα τα σημεία ελέγχου
  - **Approximation -** η καμπύλη δεν περνάει από όλα τα σημεία ελέγχου



Convex Hulls

- Το όριο που σχηματίζεται από το σύνολο των σημείων ελέγχου για μια καμπύλη είναι γνωστά ως **convex hull**



Bézier Spline Curves

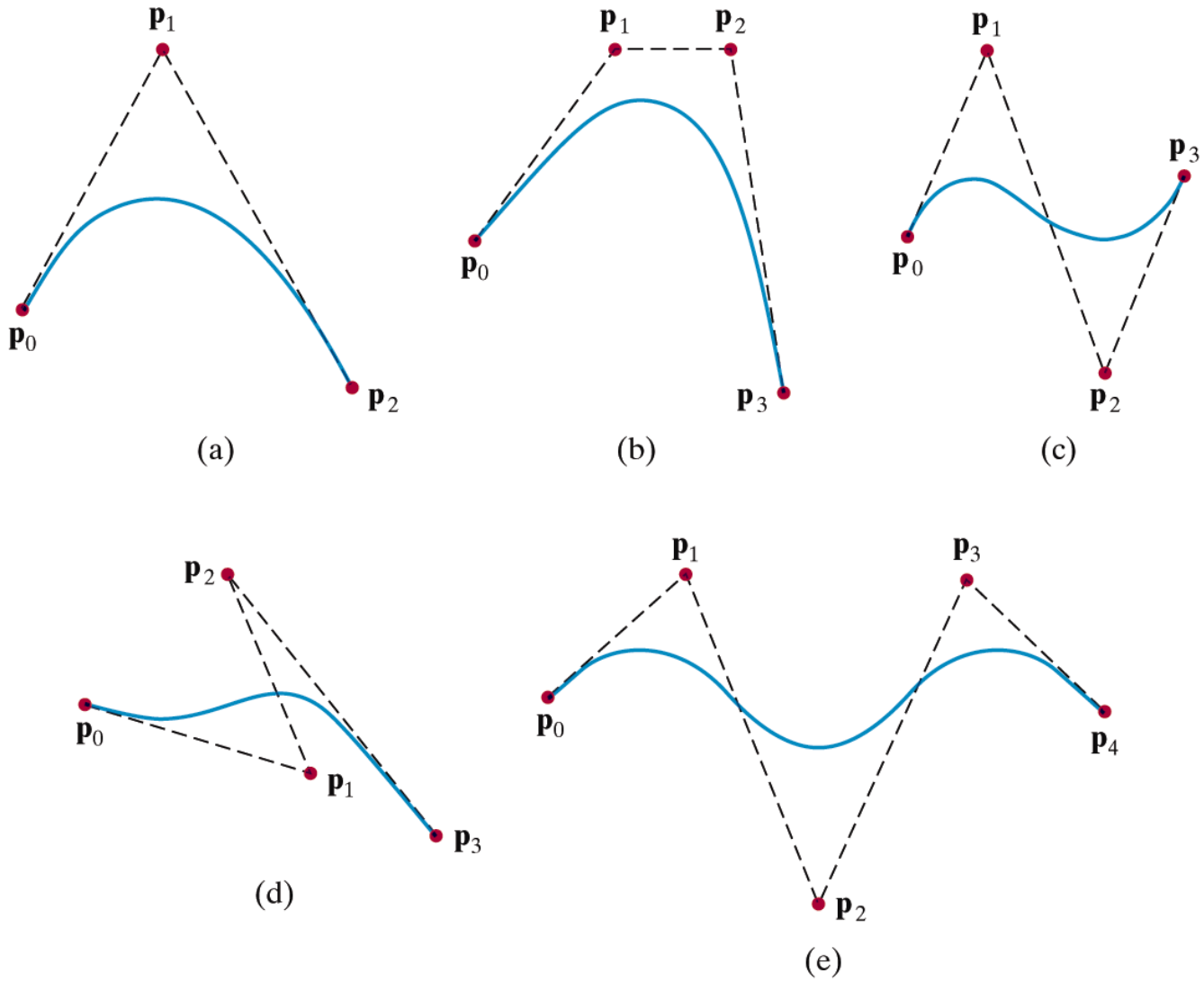
- Η πιο γνωστή μέθοδος είναι αυτή που υλοποίησε ο μηχανικός Pierre Bézier για το σχεδιασμό των αυτοκινήτων Renault
- Μια καμπύλη Bézier μπορεί να εφαρμοστεί σε οποιοδήποτε αριθμό από σημεία, αν και συνήθως χρησιμοποιούνται 4
- Έστω $n+1$ σημεία ελέγχου $p_k=(x_k, y_k, z_k)$ όπου k είναι μεταξύ του 0 και n
- Η συντεταγμένες του μονοπατιού της καμπύλης από το διάνυσμα p_0 στο p_n δίνεται από την εξίσωση

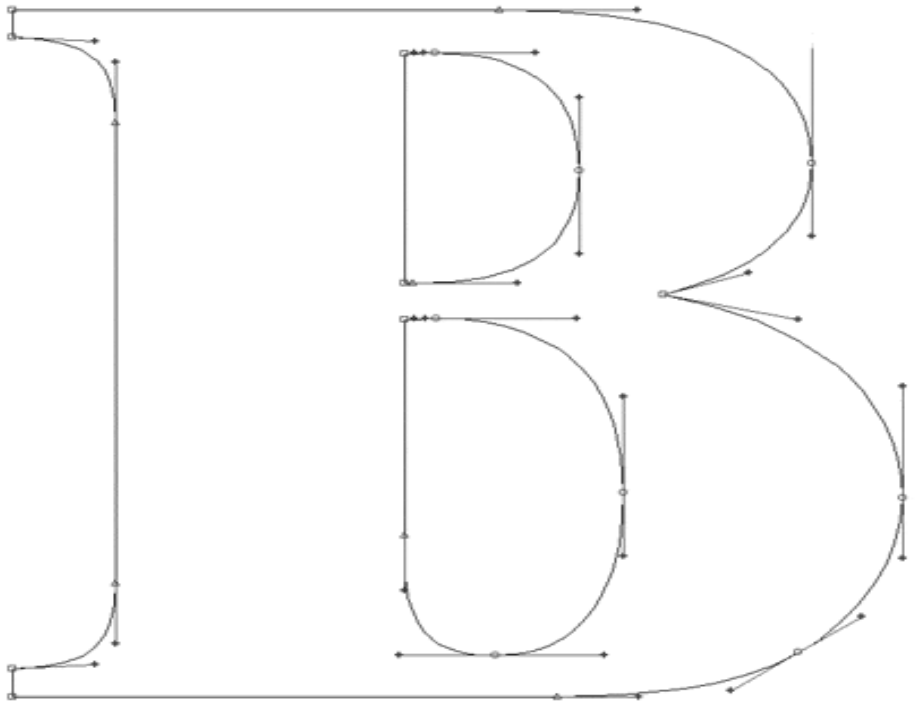
$$P(u) = \sum_{k=0}^{n} p_k BEZ_{k,n}(u), \qquad 0 \le u \le 1$$

$$BEZ_{k,n}(u) = C(n,k)u^k(1-u)^{n-k}$$

$$C(n,k) = \frac{n!}{k!(n-k)!}$$ binomial coefficients

Bézier Spline Curves



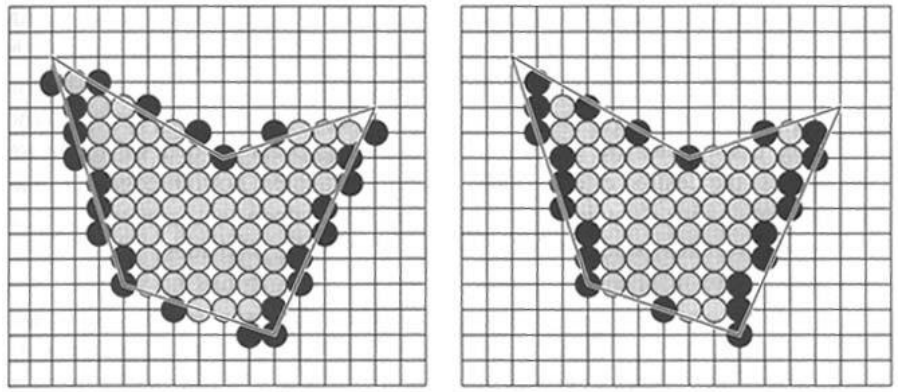


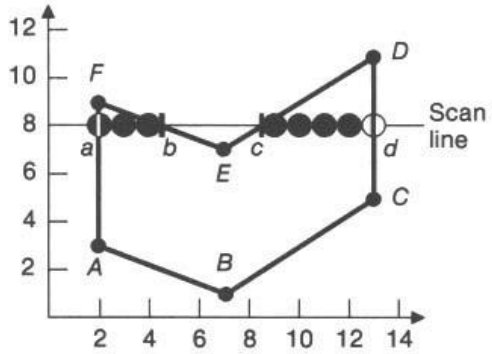
Bezier splines are widely used (Adobe, Microsoft) for font definition

Bézier Spline Curves

- Γιατί δεν προτιμάμε την χρήση καμπυλών, είτε από απλά σχήματα (κύκλο), είτε από σύνθετα σχήματα (καμπύλες Bezier);

Πολύγωνα





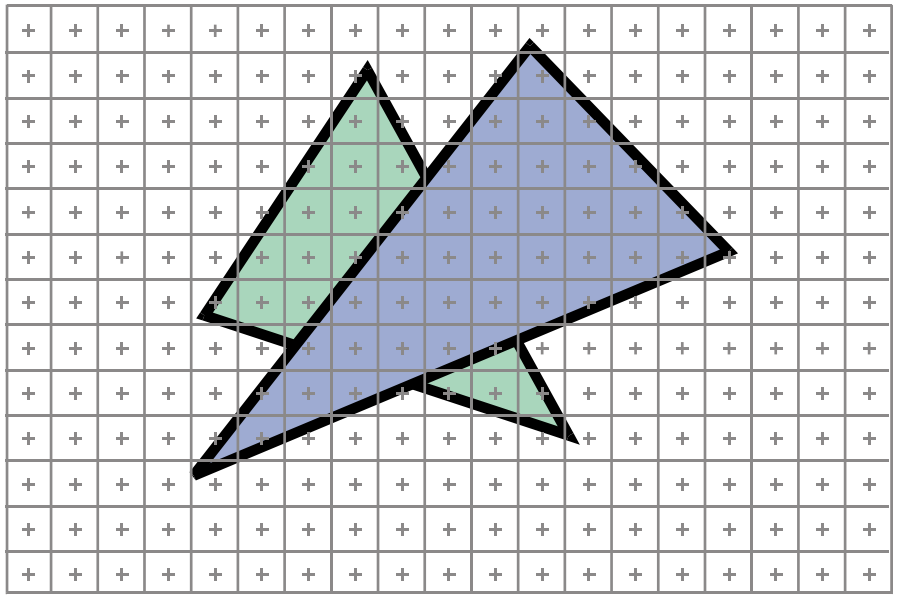
Γέμισμα Πολυγώνων

- Μάθαμε πως να σχεδιάζουμε γραμμές και κύκλους
- Πως όμως θα σχεδιάσουμε το πολύγωνα;
- Μπορούμε να χρησιμοποιήσουμε ένα κλιμακωτό αλγόριθμο γνωστό ως **scan-line algorithm**

Rasterisation (or **rasterization**) is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots)

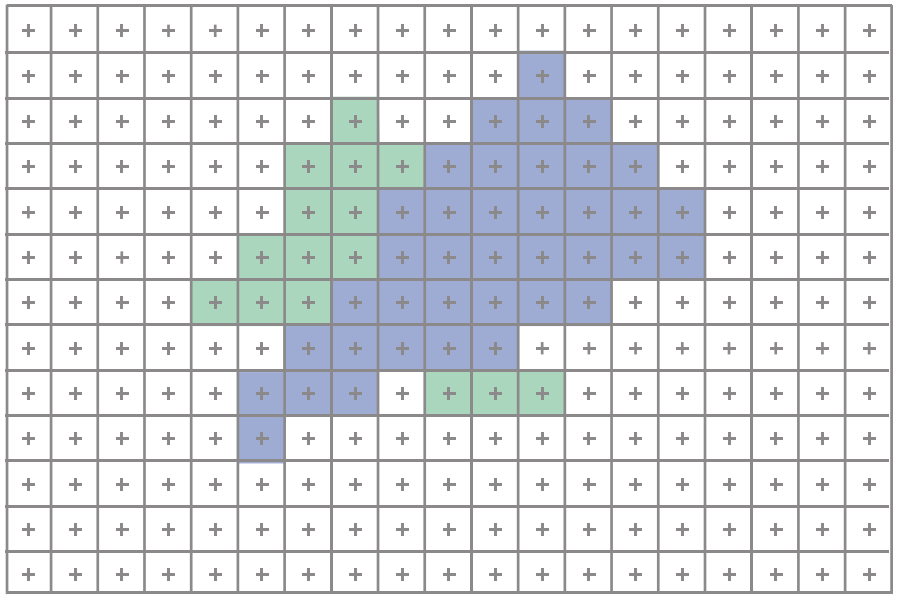
2Δ Σάρωση (Scan Conversion)

- Τα αρχέγονα σχήματα (primitives) είναι συνεχείς – η οθόνη είναι διακριτή (discrete)



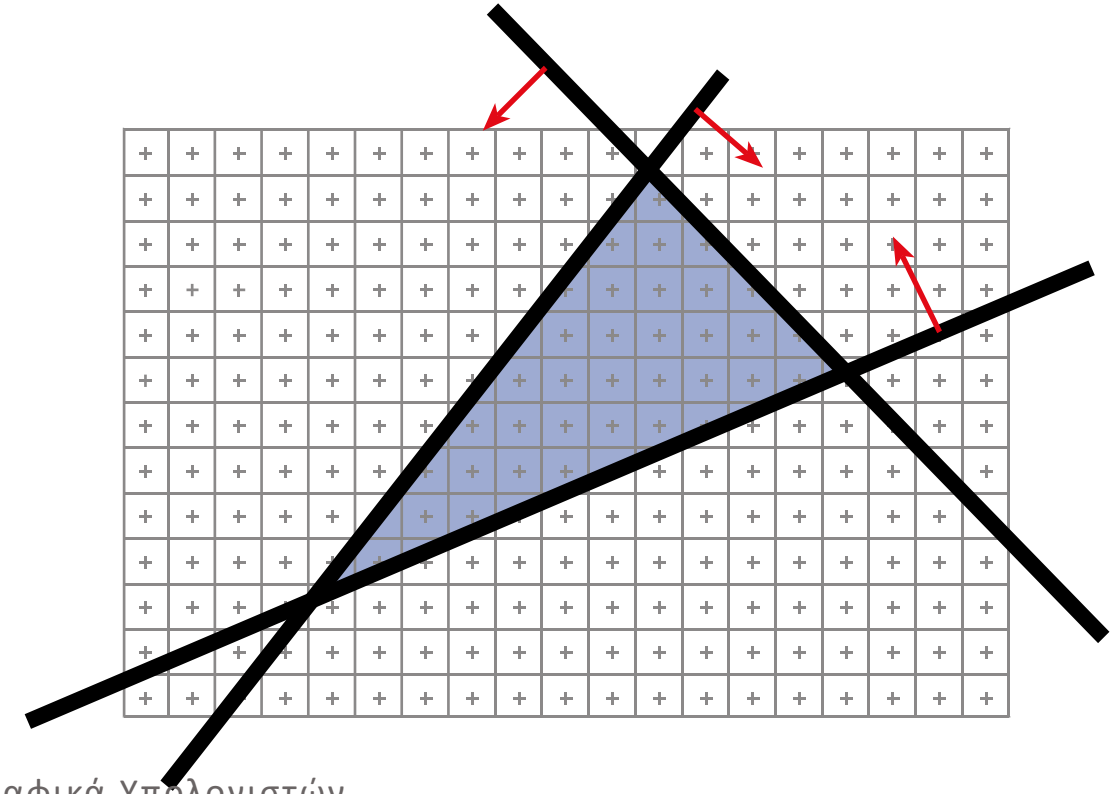
2Δ Σάρωση

- Λύση: υπολογίζουμε διακριτά με προσέγγιση
- Σάρωση: οι αλγόριθμοι για αποδοτική δημιουργία των δειγμάτων περιλαμβάνουν αυτή την προσέγγιση



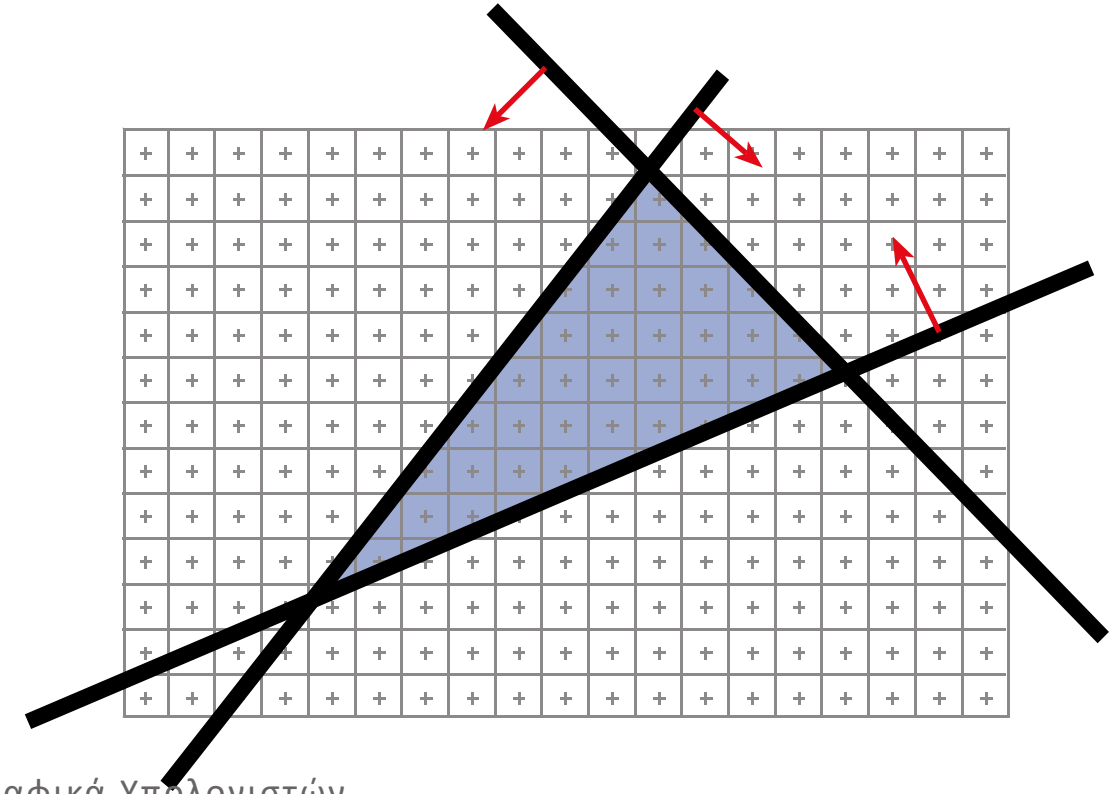
Brute force solution for triangles

- ?



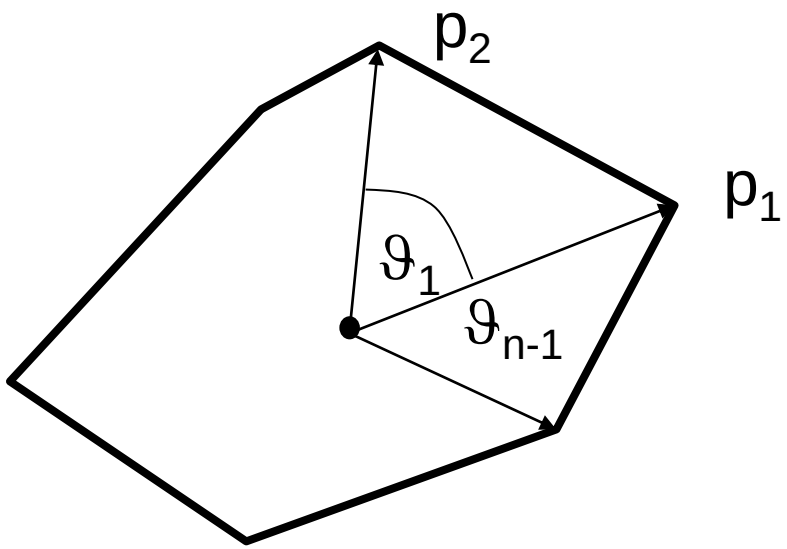
Brute force solution για τρίγωνα

- Για κάθε pixel
  - Κοιτάζουμε αν είναι **μέσα** στο τρίγωνο



Γιατί τρίγωνα;

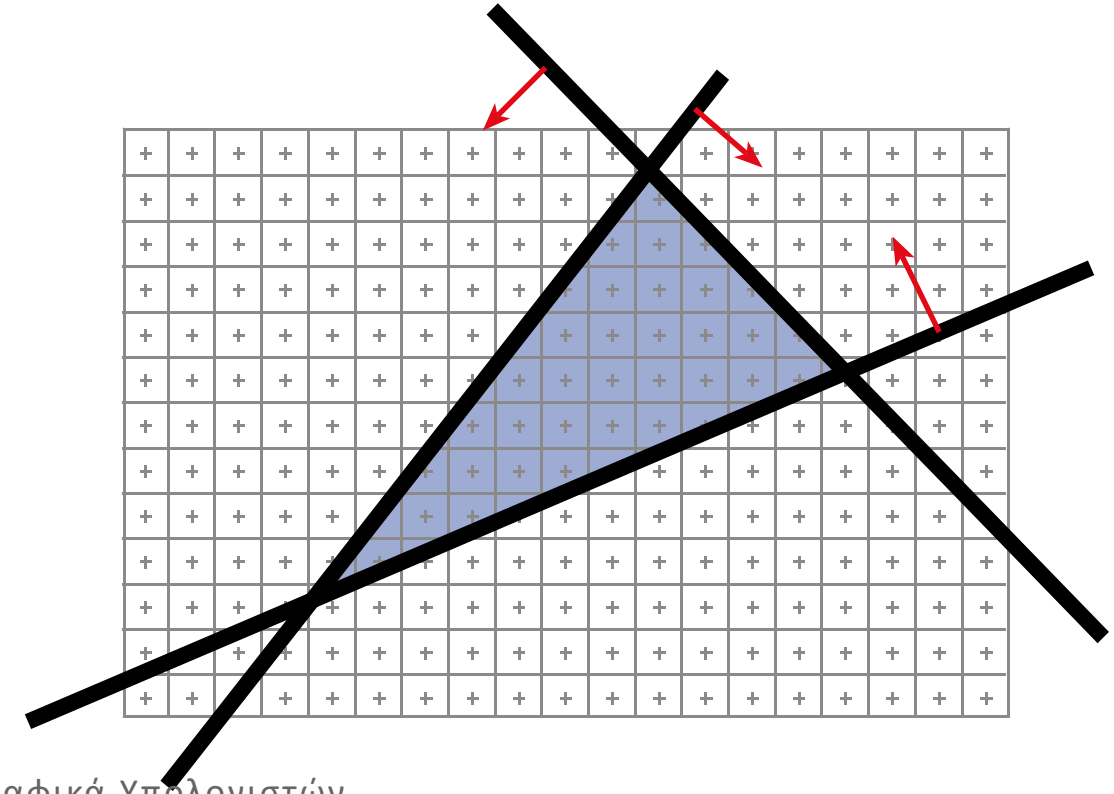
- Σημείο σε ένα πολύγωνο θα μας δώσει τρίγωνα



$$\sum_{i=1}^{n} \vartheta_i = 2\Pi$$

Brute force solution for triangles

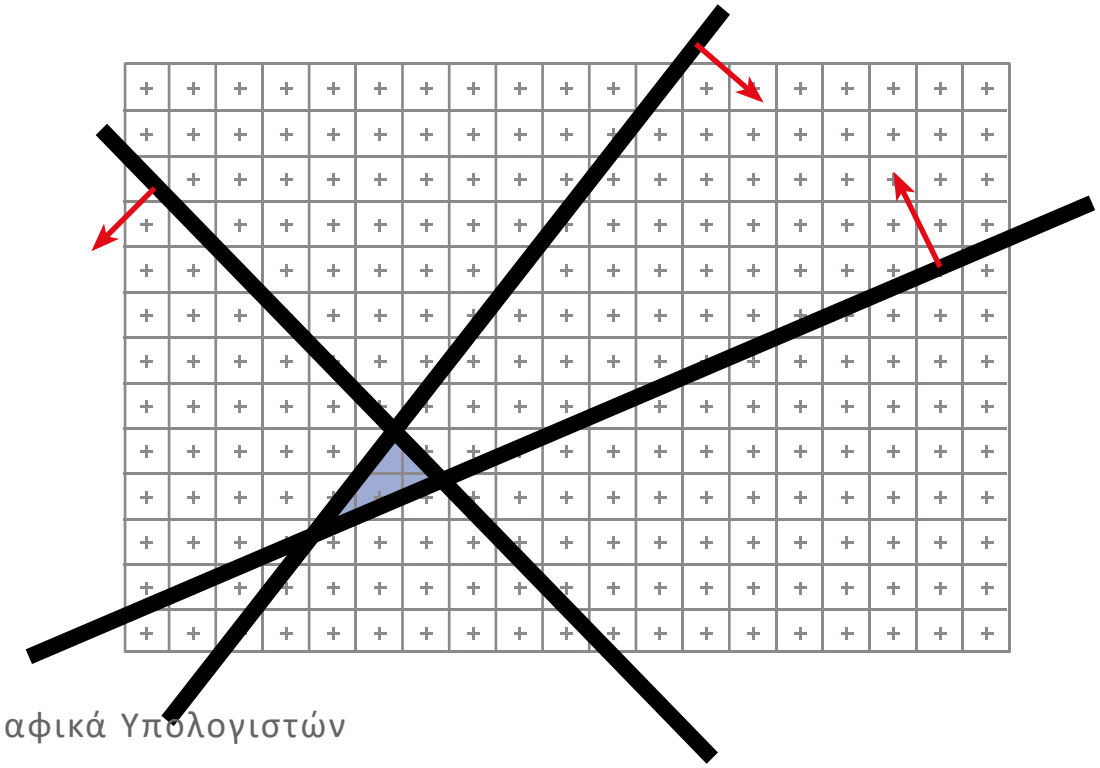
- Για κάθε πίξελ
  - Κοιτάζουμε αν είναι μέσα στο τρίγωνο



Πρόβλημα?

Brute force solution for triangles

- Για κάθε πίξελ
  - Κοιτάζουμε αν είναι μέσα στο τρίγωνο

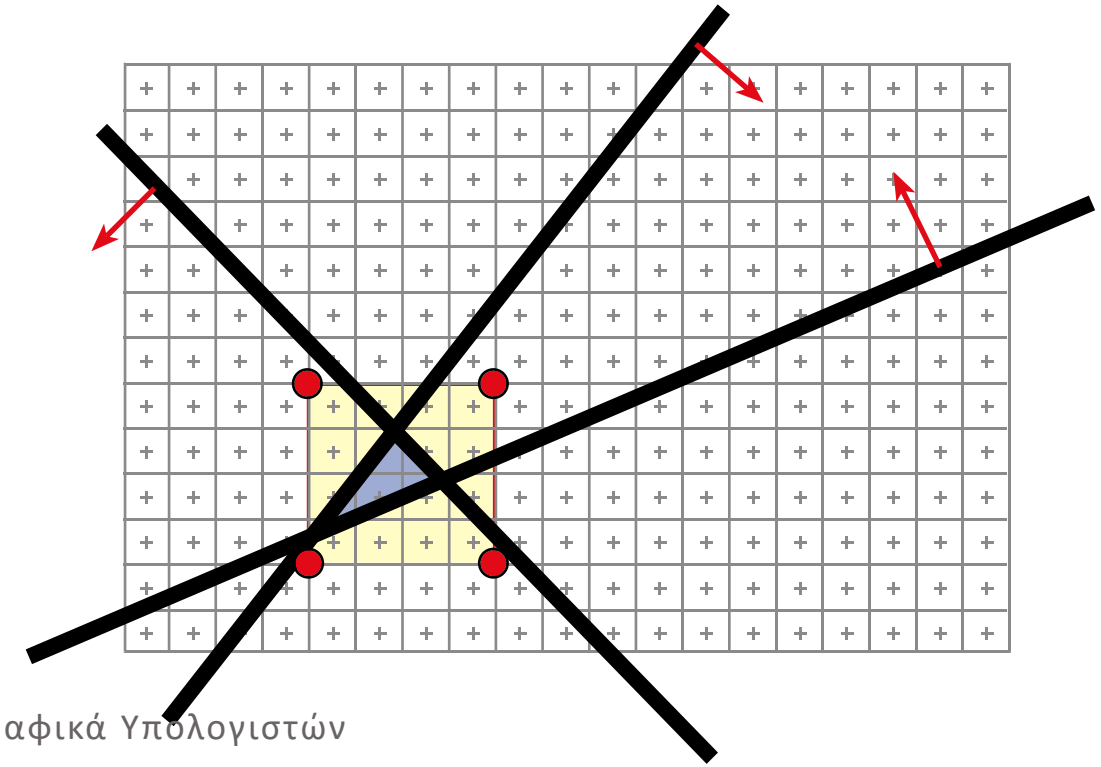


Πρόβλημα?

Αν το τρίγωνο είναι μικρό, κάνουμε πολλούς αχρείαστους υπολογισμούς

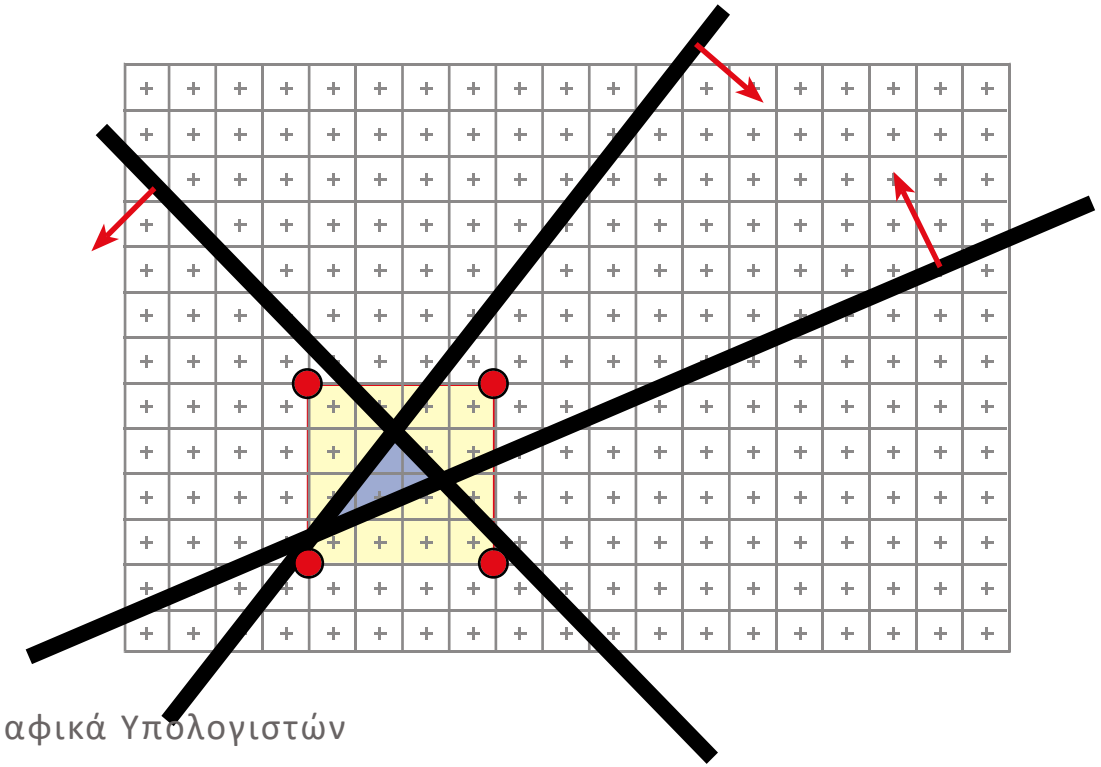
Brute force solution for triangles

- Βελτιστοποίηση:
  - Κοιτάζουμε μόνο τα πίξελς που είναι μέσα στο περιβάλλον κουτί του τριγώνου
  - Πως βρίσκουμε το περιβάλλον κουτί;



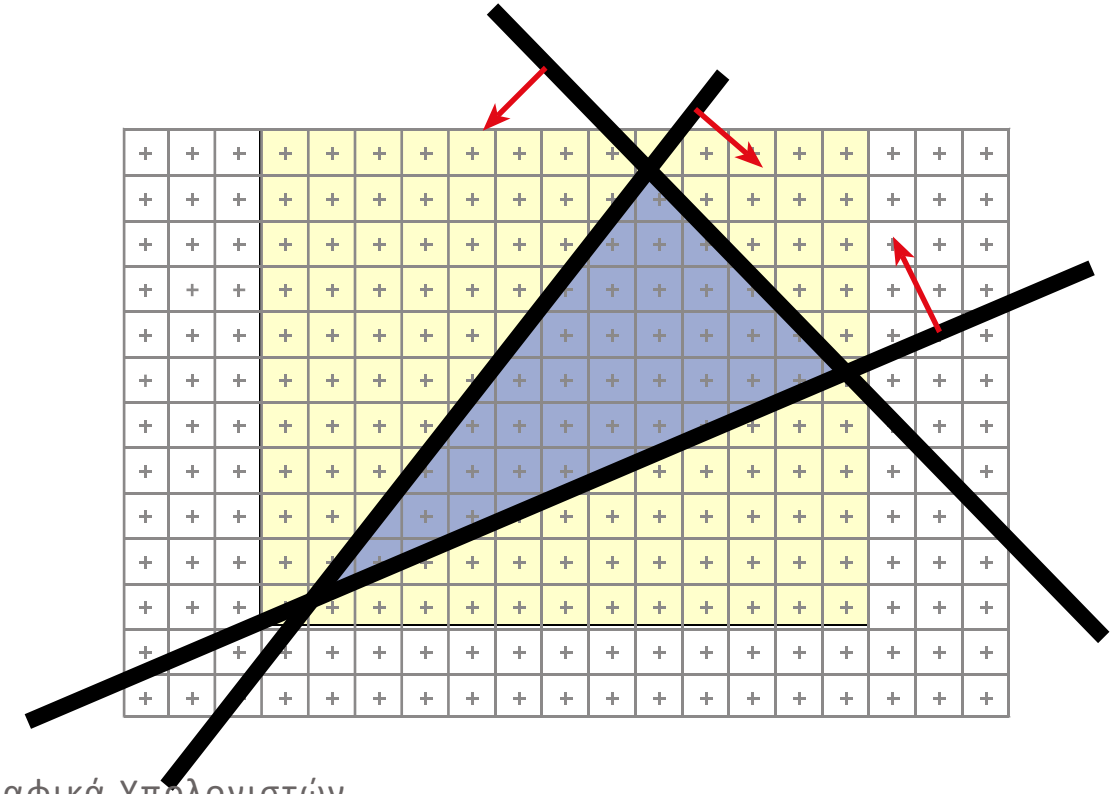
Brute force solution for triangles

- Βελτιστοποίηση:
  - Κοιτάζουμε μόνο τα πίξελς που είναι μέσα στο περιβάλλον κουτί του τριγώνου
  - Με τα Xmin, Xmax, Ymin, Ymax των κορυφών του



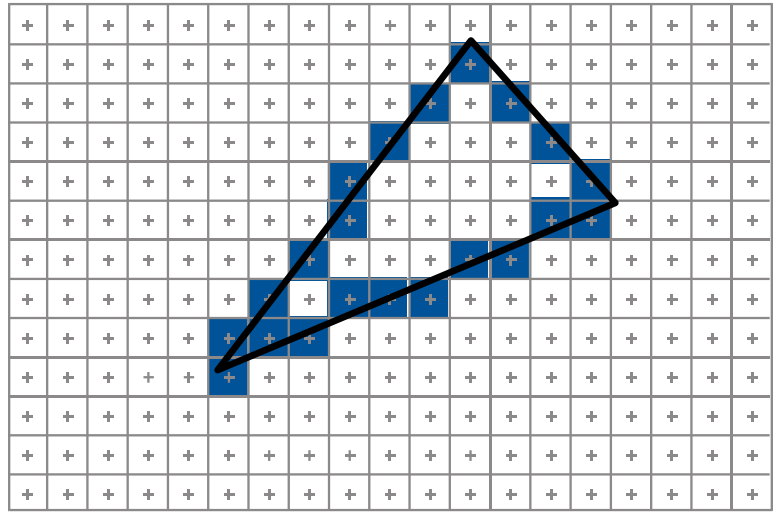
Μπορούμε να κάνουμε κάτι καλύτερο;

- Αν τα τρίγωνα είναι μεγάλα, πάλι έχουμε πολλούς αχρείαστους υπολογισμούς
- Τι μπορούμε να κάνουμε;

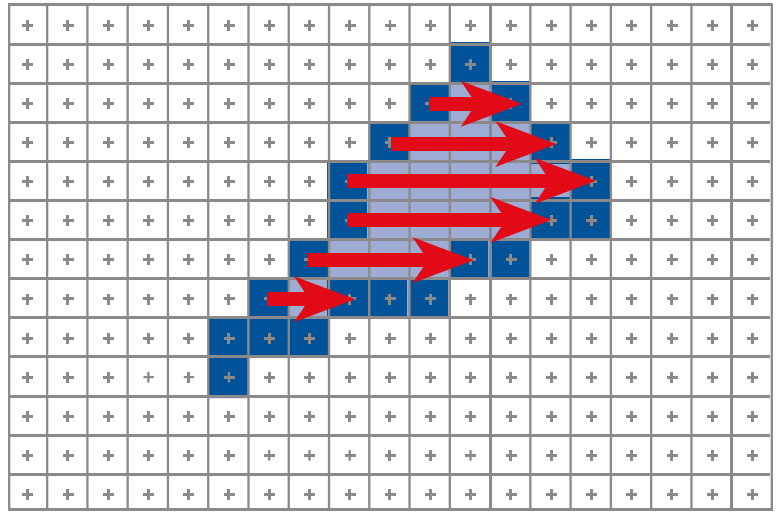


Scan-Line Polygon Fill Algorithm

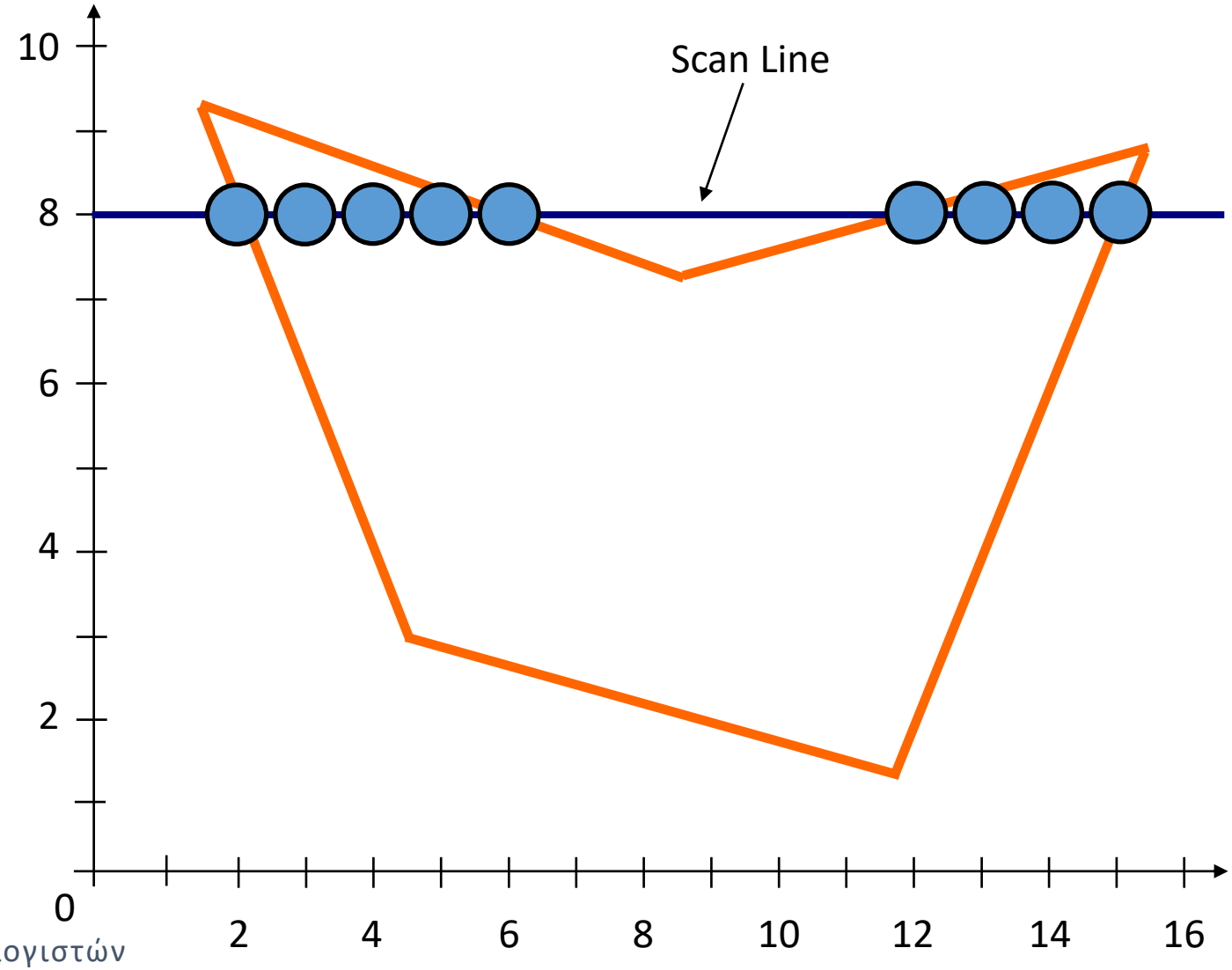
- Χρησιμοποιούμε line rasterization και μετά γέμισμα ανά γραμμή
- Υπολογίζουμε τα εικονοστοιχεία (pixels) στα όρια



- Γεμίζουμε τα κενά γραμμή προς γραμμή (spans)



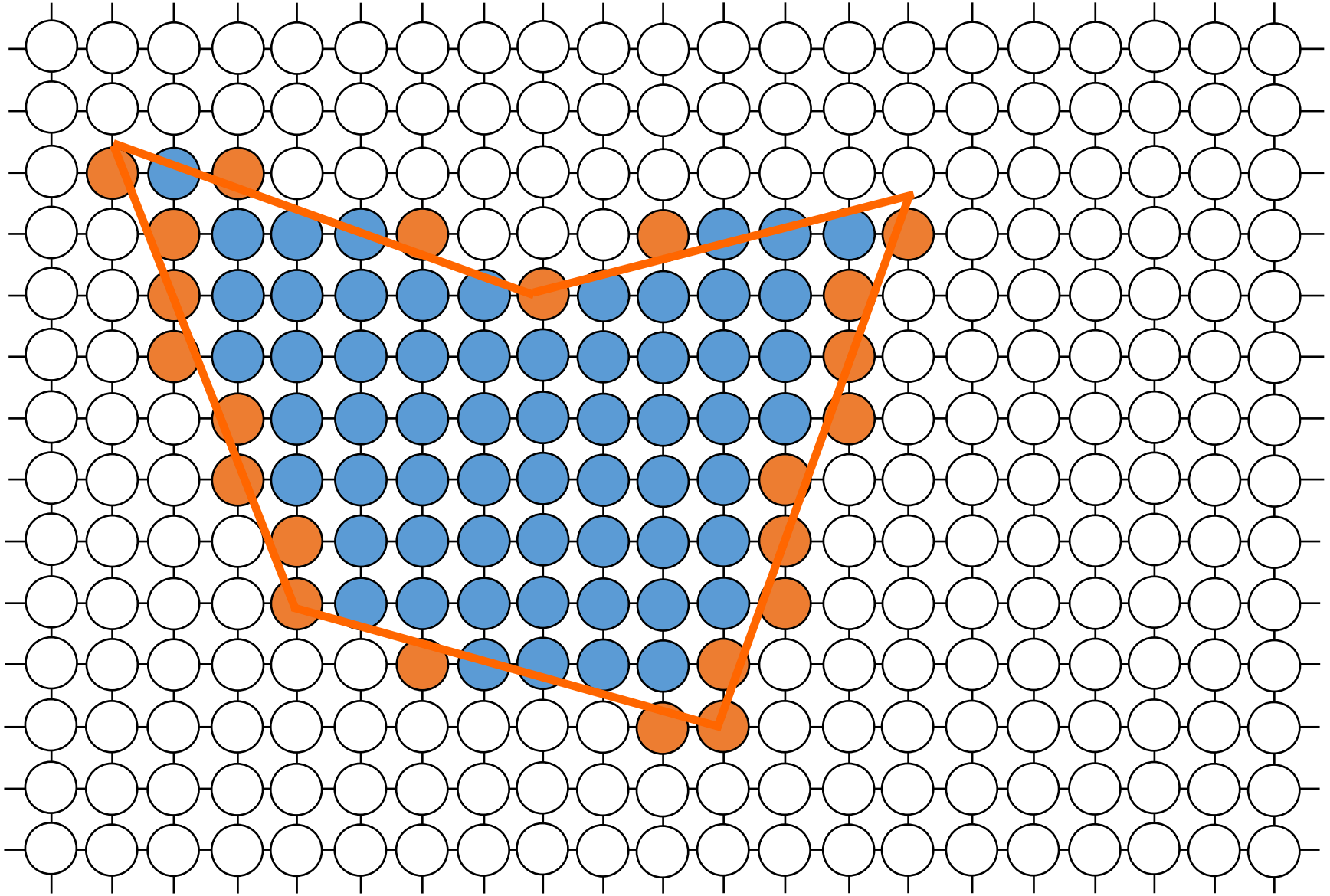
Scan-Line Polygon Fill Algorithm



Scan-Line Polygon Fill Algorithm

- Ο βασικός scan-line algorithm έχει ως εξής:
  - Βρείτε τις διασταυρώσεις της γραμμής σάρωσης με όλα τα άκρα του πολυγώνου
  - Ταξινομήστε τις διασταυρώσεις αυξάνοντας τις συντεταγμένες του x άξονα
  - Συμπληρώστε όλα τα pixel ανάμεσα σε ζεύγη διασταυρώσεων που βρίσκονται στο εσωτερικό του πολυγώνου

Scan-Line Polygon Fill Algorithm

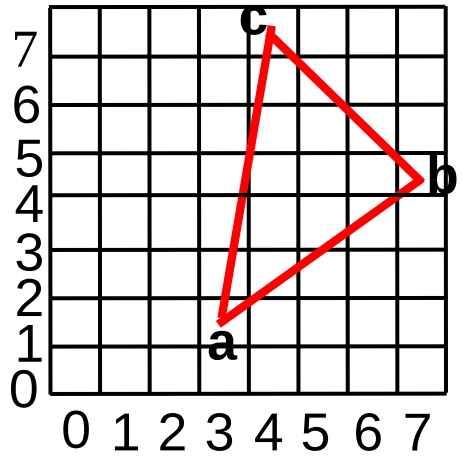


Σχεδίαση γραμμών

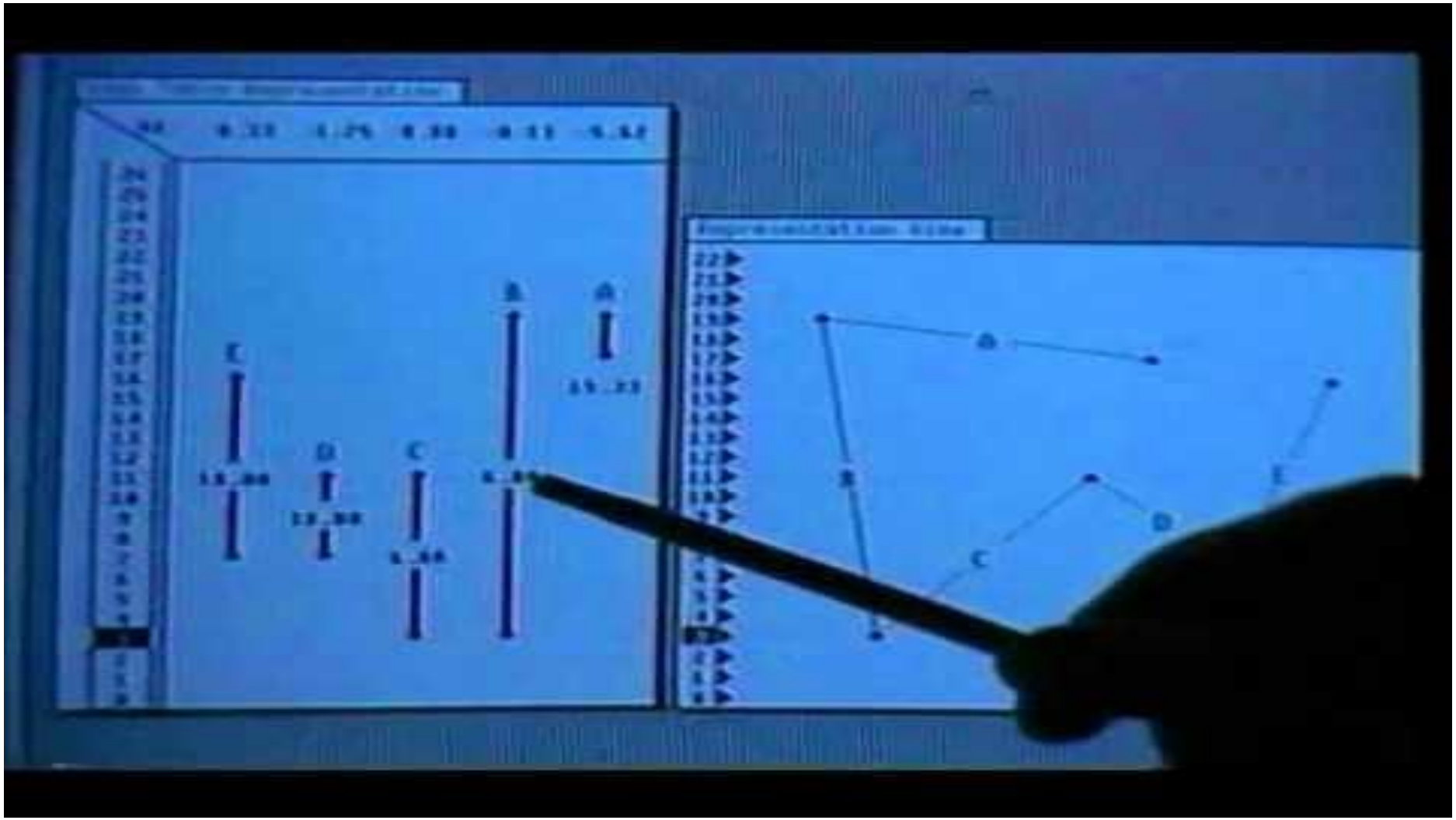
- Έχουμε εξετάσει την ιδέα της σάρωσης γραμμών
- Το βασικό χαρακτηριστικό είναι ότι πρέπει να είναι **γρήγοροι**
- Για τις γραμμές έχουμε τους αλγόριθμους DDA ή/και Bresenham
- Για τους κύκλους έχουμε τον mid-point algorithm

Σάρωση Τριγώνου

```
void scanTriangle(Triangle T, Color rgba) {
        for each edge
                compute (y2, x1, dx/dy)
        for each scanline at y
                for the current edge pair (L, R) {
                        for (int x = xL; x <= xR; x++)
                                SetPixel(x, y, rgba);
                xL += dxL/dyL;
                xR += dxR/dyR;
        }
}
```



Demo



Demo: https://youtu.be/GXi32vnA-2A