

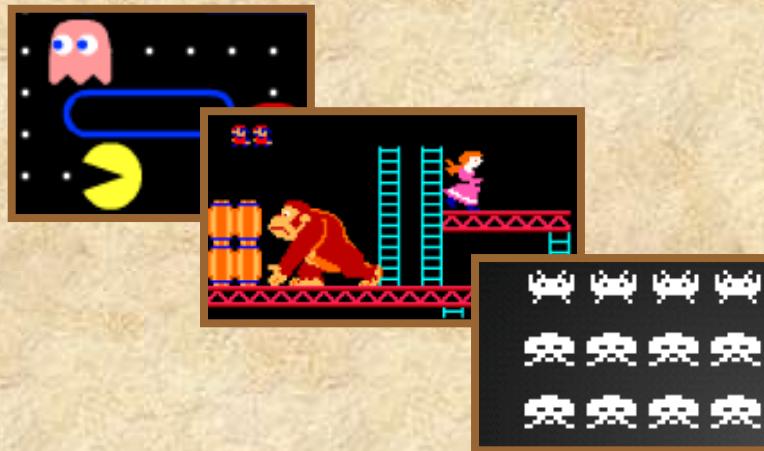


**HY454 : ΑΝΑΠΤΥΞΗ ΕΞΥΠΝΩΝ ΔΙΕΠΑΦΩΝ ΚΑΙ
ΠΑΙΧΝΙΔΙΩΝ**

**ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ**



**ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης**



ΑΝΑΠΤΥΞΗ ΠΑΙΧΝΙΔΙΩΝ, Διάλεξη 4η



Περιεχόμενα

- **Color key and masking**
- Color modulation / tinting
- Bounding boxes and collision detection
- Display frequency and vertical retrace synchronization
- Double buffering, page flipping, triple buffering



Color key and masking (1/3)

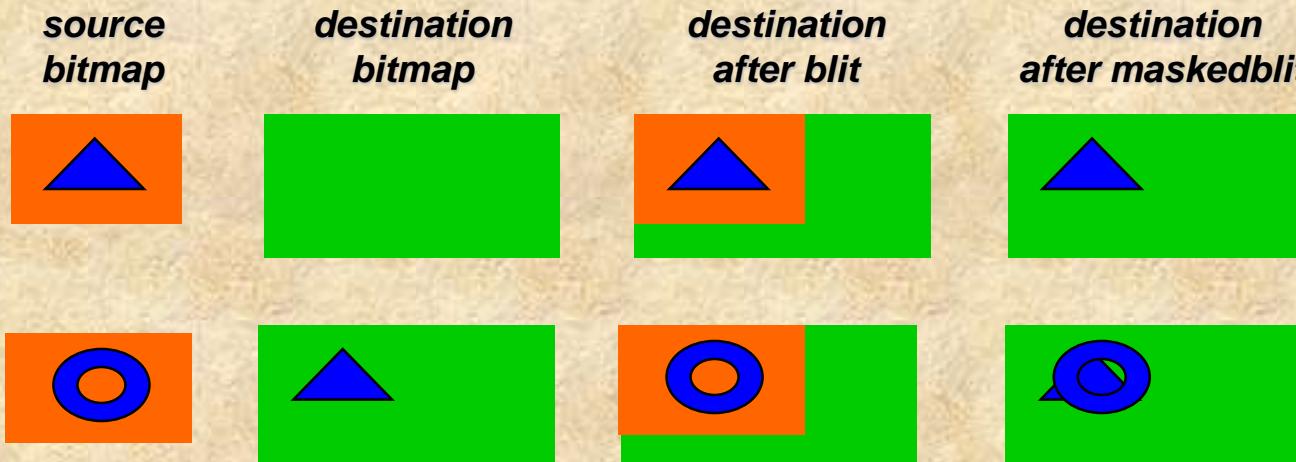
- Στα blit operations είδαμε ότι γίνεται ουσιαστικά copy του κάθε source pixel στο destination pixel χωρίς «διακρίσεις»
 - εκτός από την ειδική περίπτωση του alpha blending που υποστηρίζει blits με διάφορες τιμές «διαφάνειας» κάνοντας μίξη των source και destination pixels
- Επιπλέον υποστηρίζεται η δυνατότητα να θεωρείται τιμή ενός συγκεκριμένου χρώματος ως διαφανής - transparent, με την εξής συμπεριφορά:
 - Εάν το source pixel έχει αυτή τη συγκεκριμένη τιμή, τότε το αντίστοιχο destination pixel δεν επηρεάζεται
 - Το χρώμα αυτό ονομάζεται color key ή transparent color ή color mask
- Η ειδική αυτή blit ξεχωρίζεται από την απλή *blit* και ονομάζεται *maskedblit*.
 - Καθώς εφαρμόζει έλεγχο του source pixel είναι αργότερη από την απλή copy blit.



Color key and masking (2/3)

```
void      SetColorKey (Color c);
Color     GetColorKey (void);
Dim      MaskedBlit (
            Bitmap src, const Rect& from,
            Bitmap dest, const Point& to
        );
```

Εντοπίστε τις αντίστοιχες συναρτήσεις στη βιβλιοθήκη Allegro / SDL



Color key and masking (3/3)

- **Προσοχή** στο γεγονός ότι το **maskedblit** είναι πιο αργό από το **blit**, καθώς πρέπει σε κάθε pixel copy να κάνει σύγκριση του source pixel με το color key
 - Εάν το source bitmap δεν έχει transparent color ή
 - Εάν το destination bitmap πρέπει να γίνει εντελώς overwritten
 - χρησιμοποιείτε πάντα την απλή blit
 - ωστόσο η maskedblit καλείται ιδιαίτερα συχνά καθώς όλα τα “sprites” - χαρακτήρες / πλάσματα του παιχνιδιού - ζωγραφίζονται σχεδόν πάντοτε με masking





Περιεχόμενα

- Color key and masking
- ***Color modulation / tinting***
- Bounding boxes and collision detection
- Display frequency and vertical retrace synchronization
- Double buffering, page flipping, triple buffering



Color modulation / tinting (1/5)

- **Colour modulation** ή **tinting** είναι ο πολλαπλασιασμός όλων pixels ενός source bitmap πριν ακριβώς προηγηθεί το blit operation με ένα color
 - Σε αντίθεση με το **BitmapTintPixels** functions που υλοποιήσαμε πριν, το source bitmap παραμένει άθικτο
- Αυτό, σε alpha blending mode, επηρεάζει και το alpha value των pixels εάν το επιθυμούμε
- Είναι ένα πολύ καλό εργαλείο το οποίο μπορεί σε συνδυασμό με animations που επηρεάζουν το modulation color να δημιουργήσει πληθώρα διαφορετικών effects

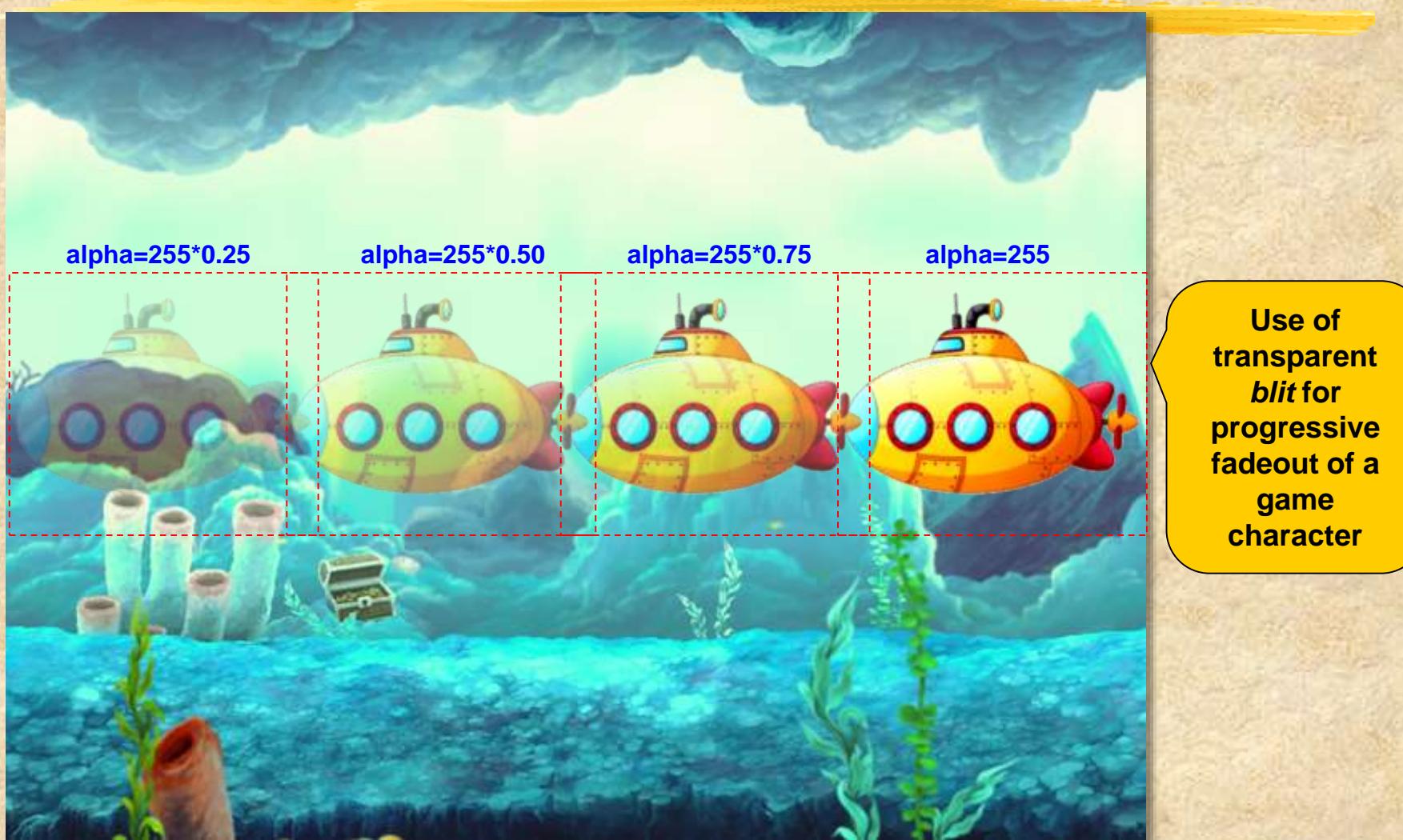


Color modulation / tinting (2/5)

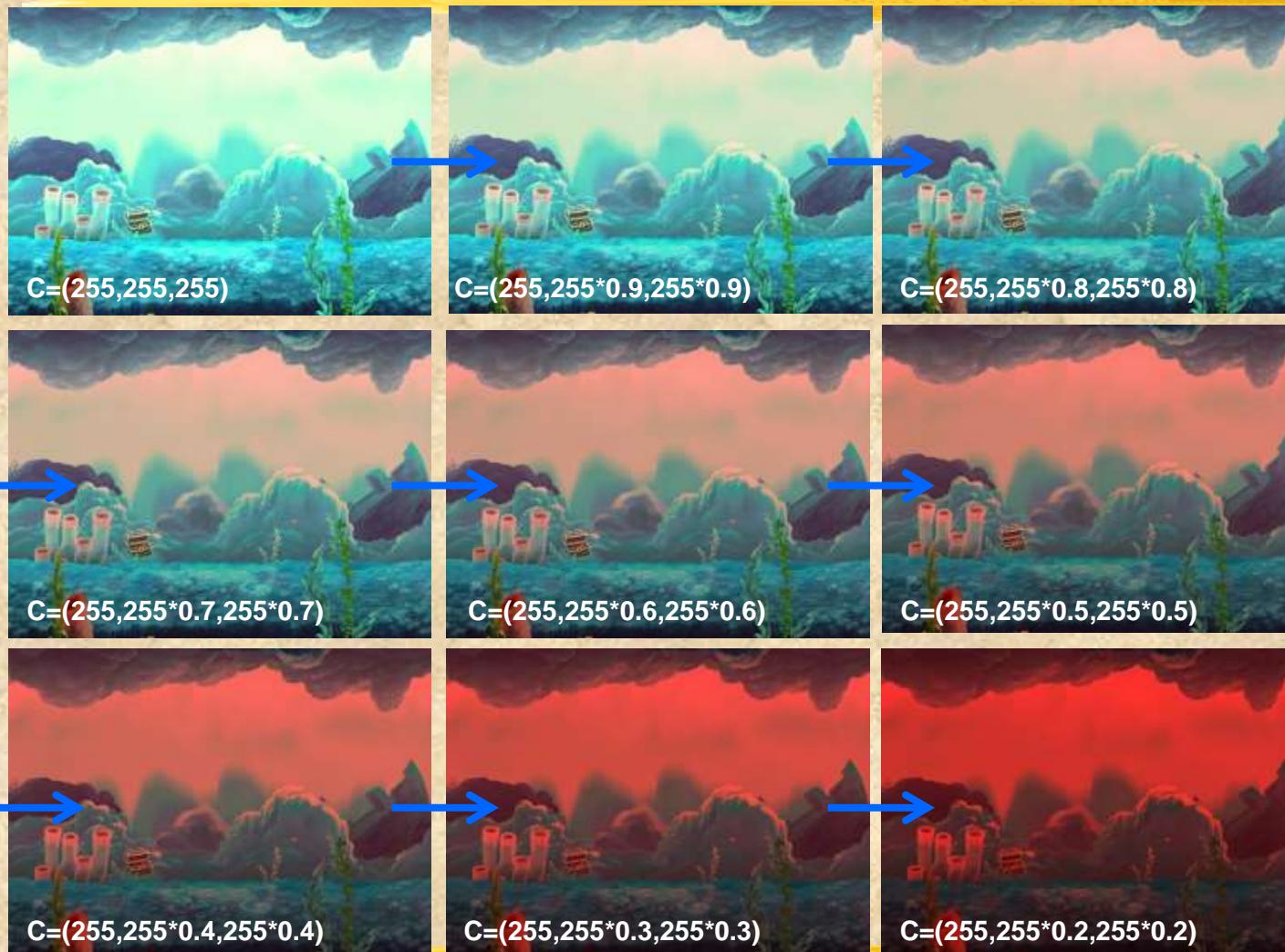
```
void BitmapBlitTinted ( ← check support in your Library
    Bitmap src, const Rect& from,
    Bitmap dest, const Point& to,
    Color modulation // src_color x modulation → target
);
<src.r * m.r, src.g * m.g,...>→dest

// emulate transparent blending
void BitmapBlitTransparent (
    Bitmap src, const Rect& from,
    Bitmap dest, const Point& to,
    RGBValue alpha
) {
    BitmapBlitTinted(
        src, from, dest, to,
        Make32 (255, 255, 255, alpha)
    );
}
```

Color modulation / tinting (3/5)



Color modulation / tinting (4/5)



Use of modulation to shift smoothly the scene background bitmap to a *reddish* look (can do it for any target color). In this case alpha is always 255.

Color modulation / tinting (5/5)



Use of modulation to progressively darken a character (for illumination it is not appropriate, since it intensifies)



Περιεχόμενα

- Color key and masking
- Color modulation / tinting
- ***Bounding boxes and collision detection***
- Display frequency and vertical retrace synchronization
- Double buffering, page flipping, triple buffering



Bounding boxes and collision detection (1/6)

- Καθώς οι χαρακτήρες του παιχνιδιού κινούνται στον δισδιάστατο χώρο του παιχνιδιού, υπάρχουν συνήθως κάποιοι κανόνες που απαιτούν υπολογισμούς όπως:
 - εάν ο χαρακτήρας «χωράει» να περάσει από κάπου στο terrain
 - εάν ο χαρακτήρας έχει συγκρουστεί (δηλ. επικαλύπτεται) με κάποιον άλλον χαρακτήρα, όπως π.χ. bullets
- Για το λόγο αυτό είναι σημαντικό να γνωρίζουμε το «ελάχιστο περικλείον ορθογώνιο» - **minimal bounding box** - του κάθε χαρακτήρα στην οθόνη
- Αλλά και να μπορούμε να υπολογίσουμε με ακρίβεια εάν συντρέχει περίπτωση επικάλυψης, καθώς ενδέχεται η σύγκριση ορθογωνίων να μην αρκεί.

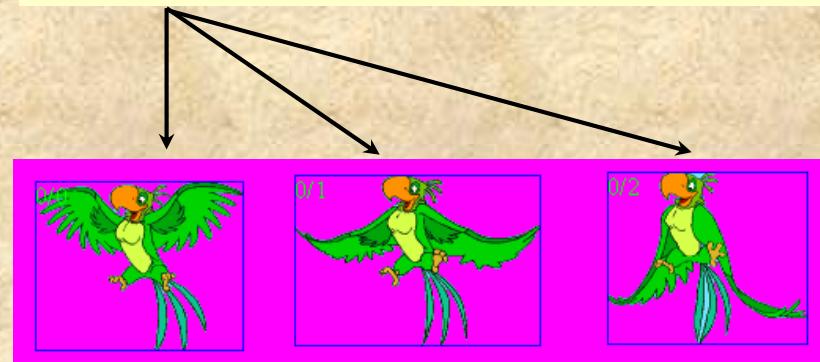


Bounding boxes and collision detection (2/6)

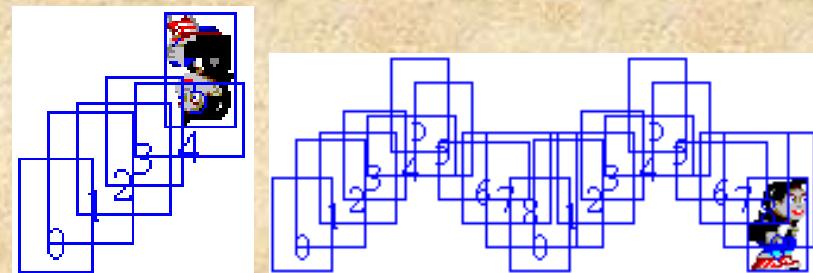
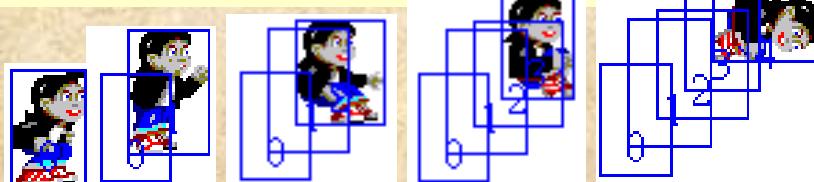
• Όταν έχουμε ένα bitmap το οποίο περιέχει τη ζωγραφιά ενός χαρακτήρα του παιχνιδιού (π.χ. σε μία συγκεκριμένη εικόνα από τις εναλλακτικές που μπορεί να έχει), τότε πρέπει αγνοώντας το transparent color να υπολογίσουμε το minimal bounding box.



• Όπως φαίνεται και στα παραδείγματα, για κάθε διαφορετική «φωτογραφία» του χαρακτήρα θα έχουμε συνήθως διαφορετικό bounding box.



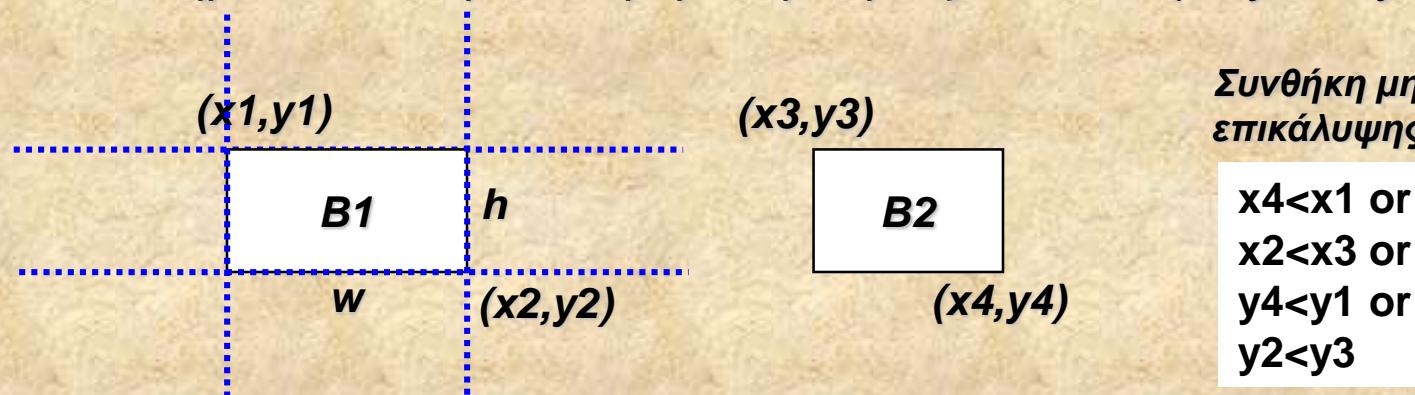
• Ο συνδυασμός της εκάστοτε θέσης (x,y) και του bounding box της εκάστοτε «εικόνας» προσδιορίζει τον ακριβή χώρο στο επίπεδο του παιχνιδιού που καταλαμβάνει κάθε χρονική στιγμή ένας χαρακτήρας.





Bounding boxes and collision detection (3/6)

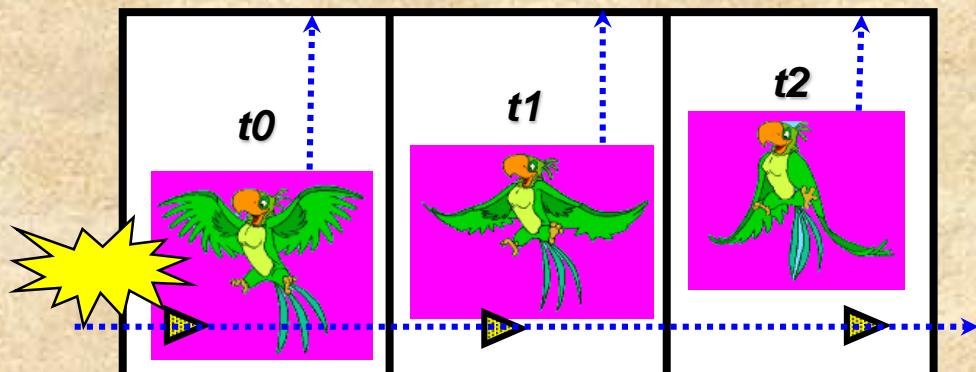
- Για να επικαλύπτονται δύο χαρακτήρες θα πρέπει τουλάχιστον να επικαλύπτονται τα bounding boxes των χαρακτήρων.
- Κατά τη μεγαλύτερη διάρκεια του παιχνιδιού στατιστικά παρατηρείται ότι συνήθως δεν υπάρχει επικάλυψη.
- Υλοποιούμε ένα γρήγορο «αρνητικό έλεγχο», δηλ. **δεν επικαλύπτονται**, σε σχέση με το **επικαλύπτονται**.
 - Αυτό ισχύει εάν το B_2 (δεύτερο box, x_3, y_3, x_4, y_4) βρίσκεται εξ ολοκλήρου: πάνω ή κάτω ή αριστερά ή δεξιά του B_1 (x_1, y_1, x_2, y_2)





Bounding boxes and collision detection (4/6)

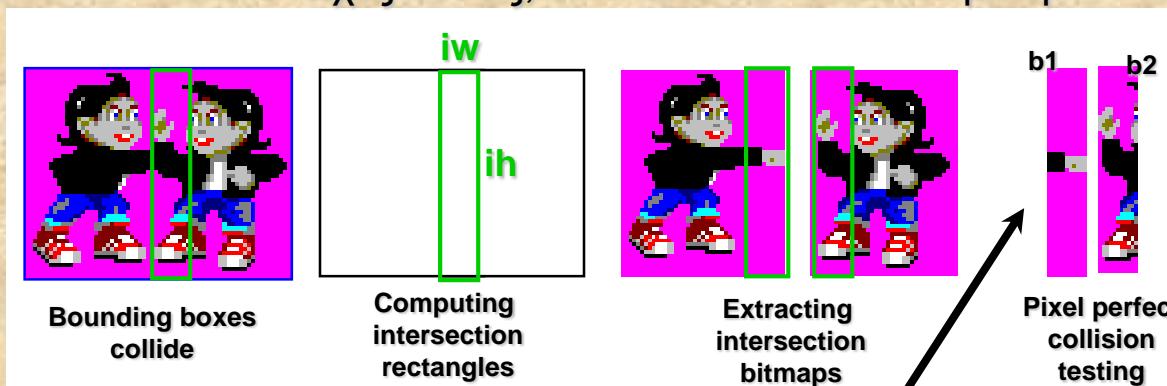
- Για να είναι γρηγορότερος ο υπολογισμός πρέπει να κρατάμε και τα δύο σημεία του bounding box, και όχι απλώς τα width και height, αλλιώς θα κάνουμε επιπλέον προσθέσεις και αφαιρέσεις.
- Η προηγούμενη συνθήκη υπολογίζεται γρηγορότερα από συνθήκη επικάλυψης που απαιτεί δύο συγκρίσεις ανά σημείο.
- ➔ To γεγονός όμως ότι τα *bounding boxes* επικαλύπτονται δεν εξασφαλίζει ότι και τα *bitmaps* στην πραγματικότητα επικαλύπτονται!



- Ο συγκεκριμένος παπαγάλος στον πραγματικό κόσμο ξεφεύγει λίγο τρομαγμένος,
- Άλλα στον κόσμο του παιχνιδιού, εάν ο προγραμματιστής απλώς βασιστεί σε bounding box collision, ήδη την χρονική στιγμή t_0 θεωρείται λανθασμένα ότι χτυπήθηκε,
- Άρα χρειαζόμαστε περισσότερη ακρίβεια στον υπολογισμό της επικάλυψης.

Bounding boxes and collision detection (5/6)

- Η προφανής λύση, η οποία μπορεί να γίνει υπολογιστικά «ασύμφορη»
 - εντοπίζουμε τα ορθογώνια της επικάλυψης (intersection rectangles), κάτι που οποίο δεν είναι προβληματικό από άποψη ταχύτητας
 - έπειτα ελέγχουμε εάν υπάρχουν δύο pixels των δύο bitmaps, στις ίδιες αντίστοιχες θέσεις, τα οποία δεν είναι διαφανή.



```
forall x:0..iw and forall y:0..ih do  
    if both pixel(b1,x,y), pixel(b2,x,y) are not transparent then  
        return true;
```

Αυτό το loop είναι πολύ αργό, καθώς κάνει πρόσβαση στο bitmap ανά pixel. Ειδικά για το collision detection που εκτελείται πιο συχνά από οτιδήποτε.



Bounding boxes and collision detection (6/6)

- Η όλη ιστορία δυσκολεύει όταν πρέπει επιπλέον του collision να κρατήσουμε και τις σχετικές συντεταγμένες των σημείων τα οποία «συγκρούονται» πρώτα
- Κάτι τέτοιο μπορεί να απαιτείται ώστε να υπολογιστεί και η αντίστοιχη «ζημιά» που θα υποστεί ένας χαρακτήρας η αντικείμενο
- Στην περίπτωση αυτή εφαρμόζουμε την τεχνική των πολλαπλών bounding boxes **για κάθε frame** τα οποία και συνήθως φέρουν μοναδικά ids
- Όταν απαιτείται τέτοια ακρίβεια μπορείτε να χρησιμοποιήσετε και bounding circles / spheres ανάλογα



• Ορίζονται ζώνες με κάθε μία να σημαίνει διαφορετική ζημιά καθώς και αντίδραση του χαρακτήρα.

• Αυτές πρέπει να ορίζονται για κάθε διαφορετική «εικόνα» του χαρακτήρα καθώς διαφέρουν.





Using bounding areas (1/7)

- Καθώς έχουμε πλέον ταχύτερες CPUs θα υλοποιήσουμε μία πιο προχωρημένη μέθοδο για collision detection που γενικεύει στα bounding boxes
- Είναι γεωμετρική (no bit buffers) και επεκτάσιμη ανάλογα με το είδος το περιγράμματος που έχει ένα sprite
- Βασίζεται σε μία προχωρημένη τεχνική που λέγεται **double dispatch**



Using bounding areas (2/7)

```
// all area classes must be defined here
class BoundingBox;
class BoundingCircle;
class BoundingPolygon;

// this super class is used by sprites
class BoundingArea {
protected:
    // required to implement the technique called double dispatching
    virtual bool Intersects (const BoundingBox& box) const = 0;
    virtual bool Intersects (const BoundingCircle& circle) const = 0;
    virtual bool Intersects (const BoundingPolygon& poly) const = 0;
public:
    virtual bool In (unsigned x, unsigned y) const = 0;
    virtual bool Intersects (const BoundingArea& area) const = 0;
    virtual BoundingArea* Clone (void) const;
    virtual ~BoundingArea(){}
};
```



Using bounding areas (3/7)

double dispatch

```
class BoundingBox : public BoundingArea {
protected:
    unsigned x1, y1, x2, y2;
public:
    virtual bool Intersects (const BoundingBox& box) const;
    virtual bool Intersects (const BoundingCircle& circle) const;
    virtual bool Intersects (const BoundingPolygon& poly) const;

    virtual bool In (unsigned x, unsigned y) const;
    virtual bool Intersects (const BoundingArea& area) const
        { return area.Intersects(*this); }
    virtual BoundingBox* Clone (void) const
        { return new BoundingBox(x1, y1, x2, y2); }
    BoundingBox (unsigned _x1, unsigned _y1, unsigned _x2, unsigned _y2) :
        x1 (_x1), y1 (_y1), x2 (_x2), y2(_y2){}
};
```



Using bounding areas (4/7)

double dispatch

```
class BoundingCircle : public BoundingArea {
protected:
    unsigned x, y, r;
public:
    virtual bool Intersects (const BoundingBox& box) const;
    virtual bool Intersects (const BoundingCircle& circle) const;
    virtual bool Intersects (const BoundingPolygon& poly) const;

    virtual bool In (unsigned x, unsigned y) const;
    virtual bool Intersects (const BoundingArea& area) const
        { return area.Intersects(*this); }
    virtual BoundingCircle* Clone (void) const
        { return new BoundingCircle(x, y, r); }
    BoundingCircle (unsigned _x, unsigned _y, unsigned _r) :
        x (_x), y (_y), r (_r){}
};
```



Using bounding areas (5/7)

```
class BoundingPolygon : public BoundingArea {
public:
    struct Point {
        unsigned x, y;
        Point (void) : x(0), y(0) {}
        Point (unsigned _x, unsigned _y) : x(_x), y(_y) {}
        Point(const Point& p) : x(p.x), y(p.y) {}
    };
    typedef std::list<Point> Polygon;
protected:
    Polygon points;
public:
    virtual bool Intersects (const BoundingBox& box) const;
    virtual bool Intersects (const BoundingCircle& circle) const;
    virtual bool Intersects (const BoundingPolygon& poly) const;

    virtual bool In (unsigned x, unsigned y) const;
    virtual bool Intersects (const BoundingArea& area) const
        { return area.Intersects(*this); }
    virtual BoundingPolygon* Clone (void) const
        { return new BoundingPolygon(points); }
    BoundingPolygon (const Polygon& _points) : points(_points) {}
};
```



Using bounding areas (6/7)

```
bool BoundingBox::Intersects (const BoundingBox& box) const {
    return !(  
        box.x2 < x1 || // at left  
        x2 < box.x1 || // at right  
        box.y2 < y1 || // above  
        y2 < box.y1 // below
    );
}

bool BoundingBox::Intersects (const BoundingCircle& circle) const
{ return circle.Intersects(*this); }

bool BoundingBox::Intersects (const BoundingPolygon& poly) const {
    // cache the polygon below inside the bounding box
    BoundingPolygon::Polygon points;
    points.push_back(BoundingPolygon::Point(x1,y1));
    points.push_back(BoundingPolygon::Point(x2,y2));
    BoundingPolygon selfPoly(points);

    return poly.Intersects(selfPoly);
}

bool BoundingBox::In (unsigned x, unsigned y) const {
    return x1 <= x && x <= x2 && y1 <= y && y <= y2;
}
```

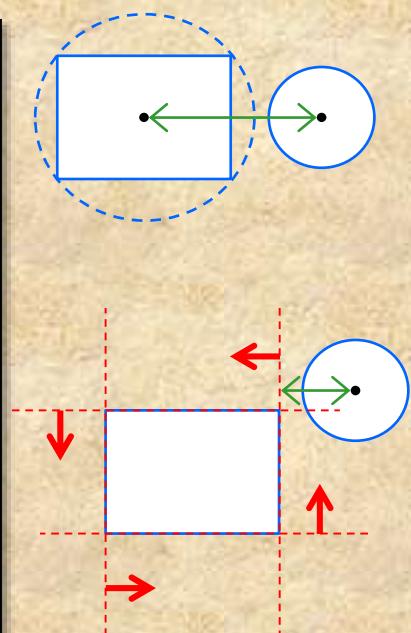
Using bounding areas (7/7)

```
bool BoundingCircle::Intersects (const BoundingBox& box) const {
    // fast failure test
    if distance of centres > (circle.radius + box.half_max_diagonal)
        return false;
    else {
        for each
            directed line of every box edge such that positive distances
            concern the half plane in which the box resides

            if the distance of the circle centre to the line
                is negative and in absolute terms larger to radius
                return false
    }
}

bool BoundingCircle::Intersects (const BoundingPolygon& poly) const {
    same than above but for N lines
}

bool BoundingCircle::In (unsigned _x, unsigned _y) const {
    return distance of (_x,_y) to centre is <= radius
}
```





Περιεχόμενα

- Color key and masking
- Color modulation / tinting
- Bounding boxes and collision detection
- *Display frequency and vertical retrace synchronization*
- Double buffering, page flipping, triple buffering

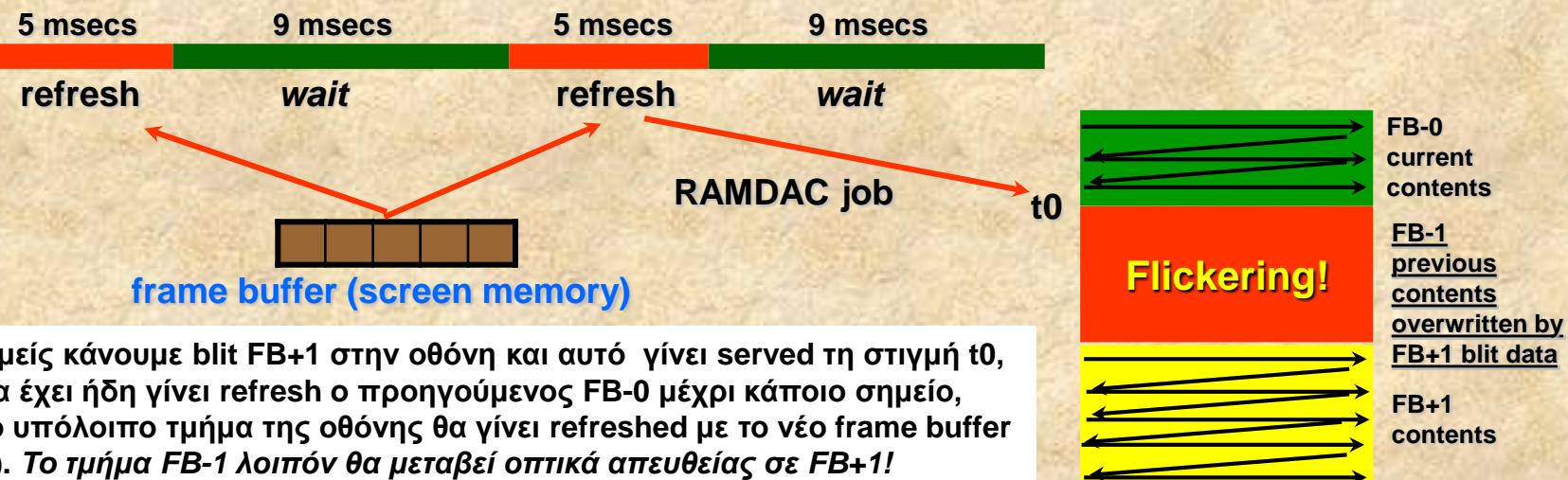


Display frequency and synchronization (1/3)

- Η αποτύπωση του frame buffer στην οθόνη γίνεται με συχνότητα που εξαρτάται τόσο από την κάρτα γραφικών όσο και από την οθόνη
 - Ονομάζεται συχνότητα ανανέωσης (display frequency) και συνήθως είναι από 60Hz έως 100 Hz
 - Στις χαμηλές συχνότητες το ανθρώπινο μάτι μπορεί να αντιλαμβάνεται την ανανέωση ιδιαίτερα εάν δεν εστιάζει απευθείας στο χώρο της οθόνης, γεγονός που δίνει την αίσθηση ασταθούς εικόνας
 - Για το λόγο αυτό οθόνες τεχνολογίας 100 Hz φαίνονται να έχουν πολύ πιο σταθερή εικόνα στο ανθρώπινο μάτι
 - ◆ Σε μία Μαγκούστα (τρωκτικό) όμως θα φαίνονται και πάλι αρκετά ασταθείς!
 - Ο χρόνος που μεσολαβεί από την στιγμή που τελειώνει μία ανανέωση, έως τη στιγμή που αρχίζει η επόμενη ονομάζεται **μεσοδιάστημα (Vertical Blanking Interval – VBlank)**, το οποίο μπορεί να κυμαίνεται ανάλογα με την ταχύτητα ανανέωσης (αυτή είναι πάντα σταθερή).
 - ◆ Το μεσοδιάστημα είναι κρίσιμο καθώς είναι ουσιαστικά πολύτιμος χρόνος που έχει το software να κάνει draw στην οθόνη

Display frequency and synchronization (2/3)

- Έτσι στα 75 Hz έχουμε 75 refreshes / sec, δηλ, μέγιστος χρόνος 14 msec συνολικά για refresh και wait time (μέχρι το επόμενο)
 - Εάν refresh time είναι 5 msec, έχουμε 9 msec «μεσοδιάστημα»
- Η σειρά με την οποία γίνεται το refreshing είναι συνήθως διαφορετικό στις παραδοσιακές CRT οθόνες από ότι στις οθόνες plasma / LCD
 - ΣΤΙΣ ΤΕΛΕΥΤΑΙΕΣ συνήθως γίνεται από αριστερά προς τα δεξιά και από πάνω προς τα κάτω, αλλά υπάρχουν και άλλες τεχνικές



Ένθετο



Tearing (or
flickering) effect



Display frequency and synchronization (3/3)

- Αυτό σημαίνει ότι πρέπει να φροντίζουμε να κάνουμε blit στον frame buffer μόνο όταν έχει τελειώσει το refresh process
 - αυτή η τακτική «παραδοσιακά» ονομάζεται **vertical retrace synchronization** ή σύντομα ως αυτούσια λειτουργία **vsync**
 - δηλ. κάνουμε blit μόνο κατά το μεσοδιάστημα
 - ◆ Προφανώς εάν το μεσοδιάστημα είναι μικρότερο του χρόνου που χρειάζεται να κάνουμε blit στην οθόνη, τότε σίγουρα ενώ το retrace έχει αρχίσει και πάλι, συνεχίζουμε (λόγω του blit) να γράφουμε στην οθόνη
 - ◆ Στις σημερινές κάρτες και υπολογιστές η ταχύτητα είναι περίπου 400-500 blits / sec ενός buffer μεγέθους οθόνης πάνω στην οθόνη (ποικίλει ανά resolution και bit depth),
 - ◆ ενώ η ταχύτητα refreshing tou frame buffer είναι αντίστοιχη αλλά και μεγαλύτερη
 - π.χ., εάν στα 75 Hz εφόσον το refreshing μπορεί και γίνεται σε 2-4 msec, υπάρχει επαρκές μεσοδιάστημα 10-12 msec για να γίνει ένα safe blit στον frame buffer
 - ο λόγος που ενώ υπάρχει δυνατότητα για 200 refreshes στην οθόνη αυτά δεν εφαρμόζονται είναι επιπλέον για να παρατείνεται ο χρόνος ζωής της οθόνης (θα καεί η οθόνη από την υπερθέρμανση!)



Περιεχόμενα

- Color key and masking
- Color modulation / tinting
- Bounding boxes and collision detection
- Display frequency and vertical retrace synchronization
- ***Double buffering, page flipping, triple buffering***



Display strategies (1/8)

- Πρόκειται για τον τρόπο με τον οποίο τελικά γίνεται η αποτύπωση της σκηνής του παιχνιδιού στην οθόνη
 - Ποτέ δεν ζωγραφίζουμε απευθείας στην οθόνη
 - ◆ αλλά πάντα χρησιμοποιούμε ένα extra bitmap διαστάσεων και bit depth ίδιου με την οθόνη, πάνω στο οποίο ουσιαστικά ζωγραφίζουμε
 - ◆ στο τέλος του rendering phase κάνουμε blit αυτό το bitmap πάνω στην οθόνη χωρίς masking
 - ◆ το bitmap αυτό λέγεται **back buffer**, ή virtual screen, ή off screen buffer
 - Συνήθως αυτός ο back buffer επιλέγεται να είναι bitmap που εδρεύει στη video RAM ώστε να έχουμε ταχύτερο blit από τον back buffer στον frame buffer
 - ◆ Ελέγξτε το γιατί μπορεί να είναι πλέον γρηγορότερο να έχετε memory back buffer παρά video back buffer



Display strategies (2/8)

■ Double buffering (1/3)

- Όταν έχουμε έναν back buffer, το rendering του game loop είναι υλοποιημένο όπως παρακάτω έχουμε double buffering:

```
extern Bitmap      GetBackBuffer (void);
extern Color       GetBackgroundColor (void);
extern Rect&      GetScreenRect (void);
extern void        Render (Bitmap target); // do game rendering to target
extern void        Vsync (void); // gfx lib function

void Flush (void) {
    BitmapClear(GetBackBuffer(), GetBackgroundColor());           // optional
    Render(GetBackBuffer());
    Vsync();
    BitmapBlit(
        GetBackBuffer(),
        GetScreenRect(),
        BitmapGetScreen(),
        Point{0,0}
    );
}
```



Display strategies (3/8)

■ *Double buffering (2/3)*

- Εάν περιμένουμε την **Vsync** είναι αδύνατο το game loop να τρέχει με ταχύτητα μεγαλύτερη της συχνότητας ανανέωσης, π.χ. 75 Hz, δηλ. 75 fps,
 - ◆ αυτό έαν σε κάθε loop iteration κάνουμε και rendering
- Άρα φαίνεται ότι η μόνη επιλογή είναι είτε να μην κάνουμε Vsync και να κάνουμε πάντα rendering
 - ◆ αυτό δυστυχώς δε δίνει καλά οπτικά αποτελέσματα ανεξαρτήτως frame rate
 - ◆ επιπλέον, εάν πετύχουμε πολύ μεγάλο frame rate, π.χ. 150, ζωγραφίζουμε χωρίς λόγο καθώς και με 100, δηλ αν παραλείψουμε 50 rendering calls, πετυχαίνουμε καλύτερο αποτέλεσμα και μπορούμε να χρησιμοποιήσουμε το χρόνο που μένει για το AI
- ή να κάνουμε VSync, αλλά να φροντίζουμε να κάνουμε rendering μόνο εάν δεν πρόκειται να περιμένουμε χρόνο μεσοδιαστήματος

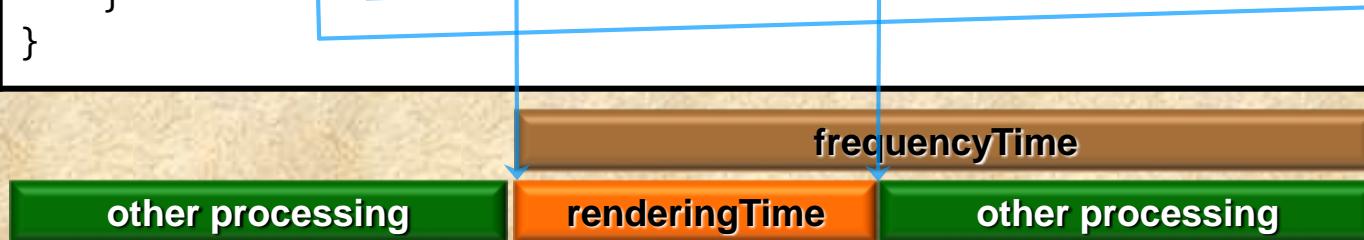
Display strategies (4/8)

Double buffering (3/3)

```
unsigned char      frequencyTime;      // e.g 14 msec για 75 Hz
unsigned int       timeToNextRendering = 0xffffffff;
extern uint64_t    CurrTime (void);    // timer in msec

void Flush2 (void) {
    if (timeToNextRendering >= CurrTime()) {
        Vsync();    // just done with refresh
        auto t = CurrTime();
        Render(GetBackBuffer()); ← also blit after this to fb
        t = CurrTime() - t; // time required to render
        timeToNextRendering = CurrTime() + (frequencyTime - t);
    }
}
```

Έτσι σκοπός είναι να ξέρω ότι κάνω Vsync προς το τέλος του refresh process, ελαχιστοποιώντας τον χρόνο αναμονής, καθώς και τον αριθμό των rendering calls.





Ένθετο

■ **Χρυσός κανόνας** ο οποίος θα σας βγάλει από αρκετούς πτονοκεφάλους

- Εάν το game loop μπορεί πάντοτε να τρέχει με ταχύτητα μεγαλύτερη του frequency ή όταν είναι πιο αργό αυτό είναι ανεκτό
 - ◆ Δεν μας απασχολεί καμία χρονοδρομολόγηση και απλά κάνουμε Vsync χωρίς να μας ενδιαφέρει το γεγονός ότι σπαταλάμε χρόνο, αφού ούτως ή άλλως δεν τον χρειαζόμαστε
- Εάν το game loop είναι αργότερο της συχνότητας της οθόνης και αυτό έχει ορατά αρνητικά αποτελέσματα τότε
 - ◆ Βελτιστοποιούμε το AI ή / και το rendering
 - ◆ Εάν δεν βελτιστοποιούνται παραπάνω και δεν μπορούμε να έχουμε καμία βελτίωση τότε
 - Είτε μειώστε την ποσότητα του AI ή την πολυπλοκότητα των σκηνών ώστε να έχουμε υπολογιστικά πιο light AI ή rendering
 - Ή κάνετε pre-caching (θα δούμε τέτοιες τεχνικές) όσο παραπάνω μπορείτε ορισμένων runtime υπολογισμών μέχρι να έχετε πραγματικά ορατή βελτίωση



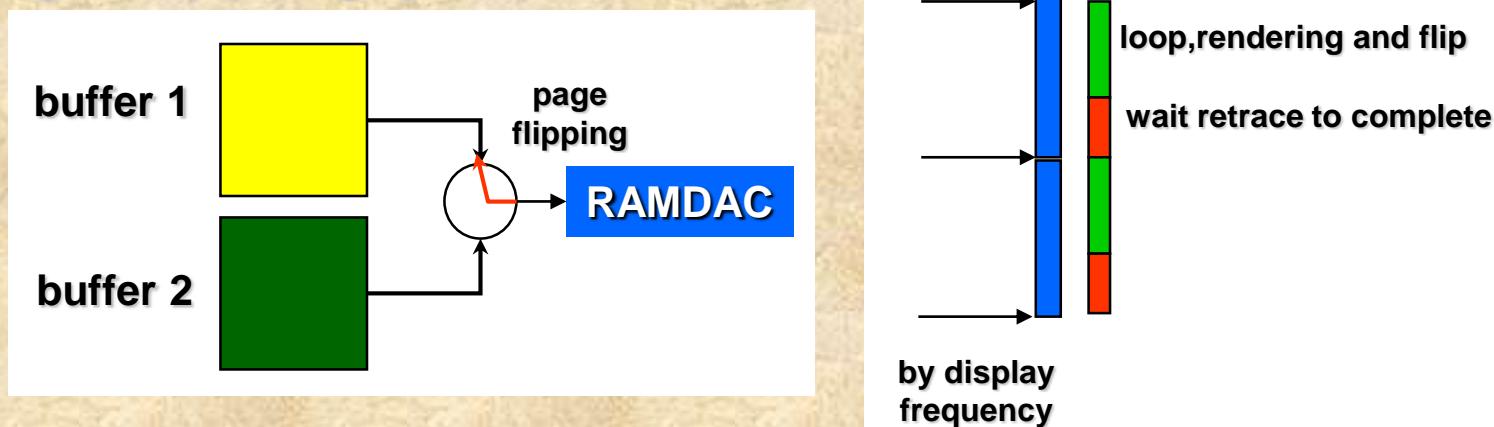
Display strategies (5/8)

■ *Page flipping (1/2)*

- Η τεχνική αυτή είναι ελαφρώς βελτιωμένη λύση του προβλήματος καθυστέρησης λόγω αναμονής της περάτωσης του display refreshing
- Πρέπει να υποστηρίζεται από το h/w και δουλεύει ως εξής:
 - ◆ Έχετε την οθόνη, δηλ. το **frame buffer** καθώς και ένα ακόμη video bitmap ίδιου τύπου που παίζει το ρόλο του **back buffer**
 - ◆ Το h/w γνωρίζει από πριν τη μνήμη που καταλαμβάνει όχι μόνο ο **frame buffer** αλλά και ο **back buffer**
 - ◆ Όταν αποτυπώσουμε στον back buffer τη σκηνή αντί να κάνουμε VSync και blit στο frame buffer, λέμε στο h/w να αλλάξει τους ρόλους των δύο bitmaps και να θεωρεί ως frame buffer τον back buffer και ως back buffer τον μέχρι πρότινος frame buffer
 - Δηλ. το h/w κάνει flip τους ρόλους τους, έτσι γλυτώνουμε το αναγκαστικό blit από τον back-buffer στο frame-buffer
 - Στο επόμενο refresh αυτομάτως τα περιεχόμενα του μέχρι πρίν back buffer θα εμφανιστούν στην οθόνη

Display strategies (6/8)

■ *Page flipping (2/2)*

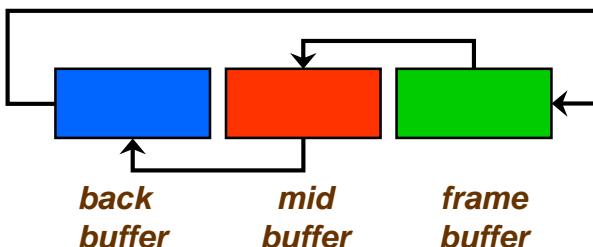


- Για να γίνει το page flipping μπορεί το h/w να περιμένει ως ότου ο παρών front buffer γίνει εντελώς refreshed.
- Επειδή όμως κάτι τέτοιο γίνεται πάντα με ελάχιστο χρόνο τη συχνότητα της οθόνης, ποτέ το game loop δεν θα ξεπερνά τη συχνότητα της οθόνης.
- Δηλ. το page flipping τείνει να συμπεριφέρεται όπως το double buffering με VSync, αφαιρώντας απλώς το blit από back buffer → frame buffer

Display strategies (7/8)

■ *Triple buffering (1/2)*

- Ότι συμβαίνει και με το page flipping αλλά με τρεις buffers αντί δύο
- Η εναλλαγή των buffers γίνεται από το h/w πάντα με μία συγκεκριμένη σειρά που λέγεται *flip chain*
- Ένας από τους τρεις είναι κάθε φορά o frame buffer, ένας o back buffer και ένας περιμένει να γίνει frame buffer (mid buffer)



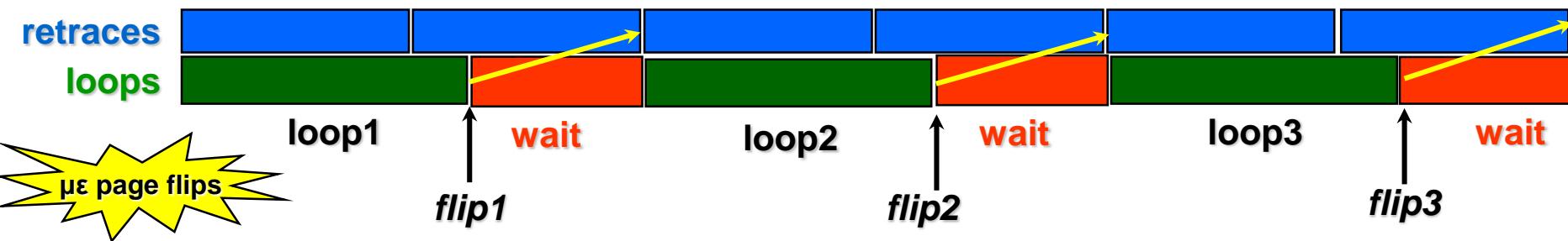
Μετά από ένα flip δρομολογείται ο back buffer να γίνει frame buffer, ενώ ο mid buffer είναι διαθέσιμος ως back buffer. Και πάλι δεν είναι δυνατόν να έχουμε loop με συνεχόμενα page flips με ταχύτητα μεγαλύτερη από αυτή της συχνότητας της οθόνης!

Display strategies (8/8)

■ Triple buffering (2/2)

- Όμως στην περίπτωση που το game loop είναι αργότερο της οθόνης, στο απλό page flipping θα περιμένουμε συχνά το επόμενο retrace.

Αυτή η συμπεριφορά μπορεί να ποικίλει
από κάρτα σε κάρτα



- Ενώ στο triple buffering δεν θα περιμένουμε ποτέ καθώς υπάρχει πάντα ένας ακόμη buffer, στο page flipping περιμένουμε να τελειώσει το επόμενο retrace και φαίνεται να τρέχουμε με fps το $\frac{1}{2}$ της συχνότητας!
- Αυτό σημαίνει ότι το triple buffering είναι πολύ καλό στην περίπτωση που έχουμε απαιτητικό game loop, δίνοντας τη δυνατότητα το frame rate να είναι το ίδιο το game loop rate.
 - ◆ Έτσι games με αρκετό AI μπορούν να τρέχουν σε 60 loops per second με rendering rate επίσης 60 fps για 60 Hz frequency



Triple buffering - ένθετο

ΚΑΝΟΝΕΣ

- Εάν τη στιγμή του flip ο mid buffer δεν έχει ακόμη γίνει served **τότε το flip περιμένει (waiting)**
 - Συμβαίνει όταν το rendering είναι ταχύτερο από το frequency
- Εάν τη στιγμή που αρχίζει το retrace ο frame buffer έχει ήδη γίνει served και εάν ο mid buffer δεν έχει γίνει served αλλάζουν ρόλους (switching)
 - Όταν δηλ. αρχίζει το current refresh και υπάρχει pending refresh
- Τη στιγμή του flip εάν ο mid buffer έχει γίνει served, τότε αλλάζει με τον back buffer (flipping)
 - Συμβαίνει όταν το refresh τρέχει γρηγορότερα από το rendering

