

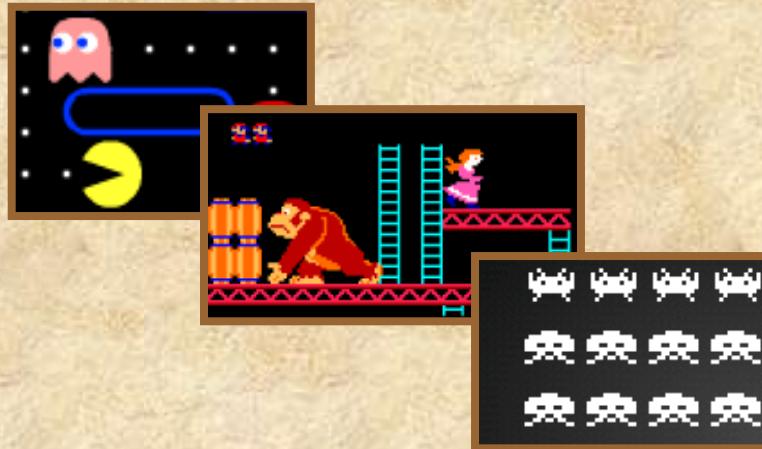


**HY454 : ΑΝΑΠΤΥΞΗ ΕΞΥΠΝΩΝ ΔΙΕΠΑΦΩΝ ΚΑΙ
ΠΑΙΧΝΙΔΙΩΝ**

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



**ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης**



ΑΝΑΠΤΥΞΗ ΠΑΙΧΝΙΔΙΩΝ, Διάλεξη 10η

Special effects



Περιεχόμενα

- **Tι είναι ειδικά effects**
- Lighting / darkening
- Transparency
- Scaling
- Rotation

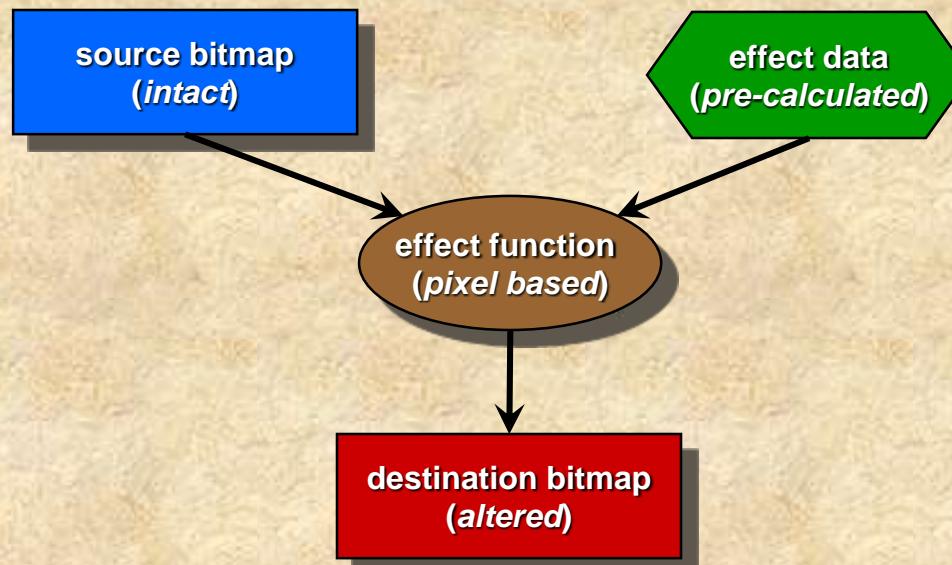


Τι είναι τα ειδικά effects (1/2)

- Πρόκειται για γραφική αποτύπωση που εξομοιώνει πραγματική ή φανταστική σκηνογραφία
 - η οποία όμως δεν μπορεί να βρίσκεται ήδη αποθηκευμένη σε προ-αποτυπωμένη (pre-rendered) μορφή στα δεδομένα του παιχνιδιού
- Τα ειδικά effects είναι αλγόριθμοι που προγραμματίζονται ώστε να πετυχαίνουν την αποτύπωση του επιθυμητού αποτελέσματος κατά την εκτέλεση σε πραγματικό χρόνο
 - Τα ειδικά effects δεν είναι τίποτε άλλο από κώδικα

Τι είναι τα ειδικά effects (2/2)

- Θα αντιμετωπίσουμε τα ειδικά effects ως επεξεργασία στο επίπεδο ενός bitmap με την ακόλουθη περίπου μορφή





Περιεχόμενα

- Τι είναι ειδικά effects
- ***Lighting / darkening***
- Transparency
- Scaling
- Rotation



Lighting / darkening (1/11)

- Πρόκειται για επεξεργασία η οποία εφαρμόζεται σε επίπεδο pixels
- Δίνουμε παρακάτω τον ορισμό μίας απλής παραλλαγής της γνωστής blit την οποία ονομάζουμε ***light blit***
 - ένα source pixel P_s γίνεται rendered στο αντίστοιχο destination pixel P_d με lighting value $L \in [0,256]$ βάσει της παρακάτω λογικής:
 - ◆ Εάν $L = 0$ (δηλ. καθόλου φως), τότε $P_d = P_s$
 - ◆ Εάν $L = 255$ (δηλ. maximum light), τότε $P_d = 255$ (λευκό)
 - ◆ Άλλιώς, $P_d = P_s + (255 - P_s) * (L / 255)$



Lighting / darkening (2/11)

- Η λογική του light blit είναι να αποτυπώσει το κάθε source pixel στο αντίστοιχο destination pixel τόσο κοντά στο λευκό χρώμα (μέγιστο φως) όσο ορίζει το light value parameter L
- Προφανώς αυτό πρέπει να εφαρμοστεί για κάθε RGB value. Έτσι η προηγούμενη λογική πρέπει να αποτυπωθεί ακριβέστερα ως εξής:
 - \forall source pixel με χρώμα C και αντίστοιχες RGB τιμές r, g, b :
 - ◆ $r_d = \text{lighten}(r, L);$
 - ◆ $g_d = \text{lighten}(g, L);$
 - ◆ $b_d = \text{lighten}(b, L);$

Lighting / darkening (3/11)

- Η υλοποίηση της συνάρτησης `lighten` είναι πάρα πολύ απλή, ωστόσο θέλει λίγο προσοχή να μην «την πατήσουμε» με την ακέραια διαίρεση

εδώ δεν «θα δείτε και πολύ φως»

```
typedef unsigned char light_t;  
RGBValue lighten (RGBValue c, light_t light)  
    { return c + (255 - c) * (light / 255); }  
RGBValue lighten (RGBValue c, light_t light)  
    { return c + ((255 - c) * light) / 255; }  
RGBValue lighten (RGBValue c, light_t light)  
    { return c + (255 - c) * (light / 255.0); }
```

εδώ όλοι οι υπολογισμοί θα δρομολογηθούν για double precision





Lighting / darkening (4/11)

- Παρατηρούμε ότι ο υπολογισμός για darkening είναι αντίστοιχος:
 - απλά αυτή τη φορά έχουμε μία παράμετρο που τη λέμε dark D αντί light
 - και με μέγιστο «σκοτάδι» προσεγγίζουμε το μαύρο χρώμα
 - ◆ Εάν $D = 0$ (δηλ. καθόλου σκοτάδι), τότε $P_d = P_s$
 - ◆ Εάν $D = 255$ (δηλ. absolute darkness), τότε $P_d = 0$ (σκότος)
 - ◆ Άλλιώς, $P_d = P_s - P_s * (D / 255)$
 - \forall source pixel με χρώμα C και αντίστοιχες RGB τιμές r, g, b :
 - ◆ $r_d = \text{darken}(r_s, D);$
 - ◆ $g_d = \text{darken}(g_s, D);$
 - ◆ $b_d = \text{darken}(b_s, D)$



Lighting / darkening (5/11)

- Και η υλοποίηση της συνάρτησης `darken` είναι πάρα πολύ απλή:

```
typedef unsigned char dark_t;  
RGBValue darken(RGBValue c, dark_t dark)  
    { return c - (c * dark) / 255.0; }
```

- Πως χρησιμοποιούμε τις απλές αυτές συναρτήσεις για να υλοποιήσουμε αντίστοιχα τις `light blit` και `dark blit`:

- `LightBlit (Bitmap src, const Rect& from, Bitmap dest, const Point& to, light_t light);`
- `DarkBlit (Bitmap src, const Rect& from, Bitmap dest, const Point& to, dark_t dark);`
- Προφανώς χρειάζεται ένας pixel-based αλγόριθμος ο οποίος να διατρέχει όλα τα source pixels και να γράφει στο destination το αντίστοιχο pixel value
 - ◆ Οι παραπάνω θεωρούμε ότι είναι υλοποιημένες να εφαρμόζουν masked lighting / darkening blit



Lighting / darkening (6/11)

```
LightBlit (Bitmap src, const Rect& from, Bitmap dest, const Point& to, light_t light) {  
    for each x : from.x to from.x + from.w-1 do  
        for each y : from.y to from.y + from.h-1 do {  
            Let C<r, g, b> = GetPixel(src, x, y);  
            if C ≠ GetColorKey() then  
                PutPixel(  
                    dest,  
                    to.x + (x - from.x),  
                    to.y + (y - from.y),  
                    MakeColor(lighten(C.r, light), lighten(C.g, light), lighten(C.b, light))  
                );  
            }  
        }  
}
```

- Προφανώς η υλοποίηση με την get / put pixel είναι απαράδεκτα αργή, συνεπώς χρειάζεται απευθείας πρόσβαση στη μνήμη των bitmaps



Lighting / darkening (7/11)

- Επιπλέον μπορούμε να παρατηρήσουμε ότι όλοι οι υπολογισμοί των συναρτήσεων `lighten` και `darken` μπορούν να γίνουν pre-cached σε δύο πίνακες των 256x256 bytes (64K):

```
static unsigned char lightingTable[256][256];

void ProducelightingTable (void) {
    for (byte light = 0; light<256; ++light)
        for (byte rgb = 0; rgb < 256; ++rgb)
            lightingTable[light][rgb] = lighten(rgb, light);
}

Let L : light value, then: auto* l = lightingTable[L];
For any C: [0..255], Lighten value = l[C]
static unsigned char darkeningTable[256][256];

void ProduceDarkeningTable (void) {
    for (byte dark = 0; dark<256; ++dark)
        for (byte rgb = 0; rgb < 256; ++rgb)
            darkeningTable[dark][rgb] = darken(rgb, dark);
}
```

προσοχή στη διάταξη
του πίνακα -
η πρώτη διάσταση
είναι πάντοτε το light
/ dark value

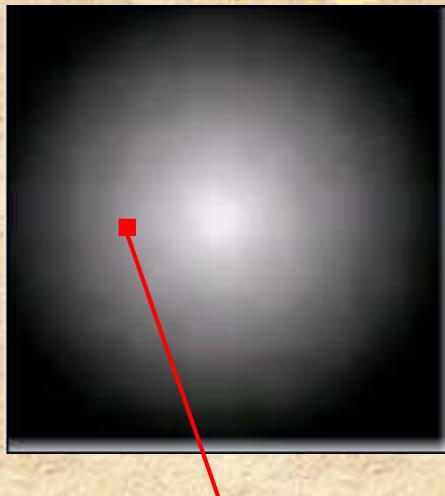


Lighting / darkening (8/11)

- Μάσκες για lighting / darkening με μεταβλητές τιμές για light / dark
 - Πέρα από το απλό blit με σταθερή τιμή στο light / dark value, σε αρκετές περιπτώσεις επιθυμούμε να έχουμε αποτύπωση ενός bitmap με μεταβλητές τιμές φωτός
 - Δηλ. Θα θέλαμε γενικά να έχουμε τη δυνατότητα να ορίζουμε διαφορετικό light / dark value για κάθε διαφορετικό pixel
 - ◆ Τέτοια πληροφορία μπορεί να αποθηκευτεί σε ένα bitmap, το οποίο προφανώς πρέπει να κατασκευαστεί γνωρίζοντας το source bitmap το οποίο θα φωτίσει / σκοτεινιάσει κατά το blit σε κάποιο destination bitmap



Lighting / darkening (9/11)



Το αριστερό Bitmap είναι μία gray «μάσκα», δηλ. ένα bitmap όπου κάθε pixel έχει ίδιες τιμές για RGB values. Άρα έχει 256 το πολύ χρώματα (αποχρώσεις του γκρι), ενώ κάθε pixel λαμβάνεται ως light / dark value.

Έστω RGB(134,134,134), τότε αντιπροσωπεύει light / dark value 134.



Η ίδια τεχνική με πολλά light blit σε διαφορετικά σημεία για effect «φωτεινής ουράς» (36 light blit)



Εδώ φαίνεται πως λαμβάνοντας ένα τμήμα από το εκτυπωμένο terrain και ξανατυπώνοντας το με light blit πετυχαίνουμε φωτισμένο terrain



Η ίδια τεχνική με μάσκα μικρότερη έντασης



Lighting / darkening (10/11)

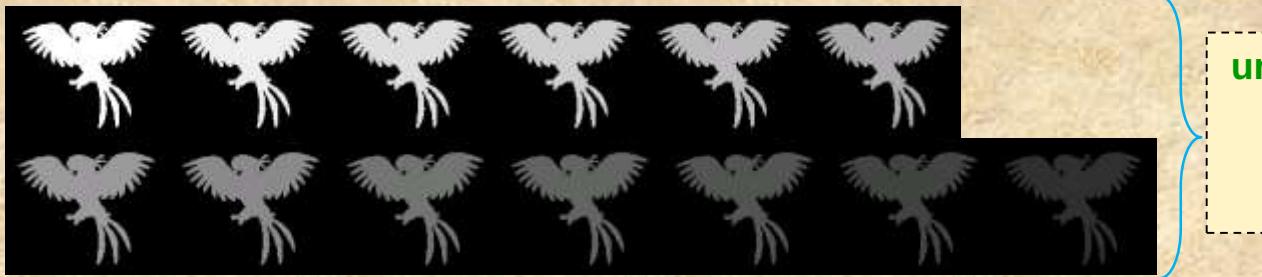
- Αλλάζει λίγο η ειδική blit ώστε αντί να έχει μία απλή παράμετρο ένα light / dark value λαμβάνει τώρα ένα light / dark bitmap mask ίδιου μεγέθους με το source bitmap

```
LightBlit (Bitmap src, const Rect& from, Bitmap dest, const Point& to, Bitmap lightMask) {  
    for each x : from.x to from.x + from.w-1 do  
        for each y : from.y to from.y + from.h-1 do {  
            Let C<r, g, b> = GetPixel(src, x, y);  
            Let light be the R value of GetPixel(lightMask, x, y);  
            if C ≠ GetColorKey() then  
                PutPixel(  
                    dest,  
                    to.x + (x - from.x),  
                    to.y + (y - from.y),  
                    MakeColor(lighten(C.r, light), lighten(C.g, light), lighten(C.b, light))  
                );  
        }  
}
```

Προφανώς δεν υπάρχει ανάγκη το lightMask να είναι bitmap, μπορεί να είναι ένας δισδιάστατος byte array

Lighting / darkening (11/11)

- Μπορείτε να κατασκευάστε utilities τα οποία δημιουργούν αυτόματα progressive light / dark masks για οποιοδήποτε frame ή και film



uniform lighting per mask,
multiple masks for
progressive lighting –
prefer light blit here



non-uniform lighting, with
progressive fade-out at frame
edges – if possible prefer post-
processed copied films here



Περιεχόμενα

- Τι είναι ειδικά effects
- Lighting / darkening
- ***Transparency***
- Scaling
- Rotation



Transparency (1/6)

- Πρόκειται για επεξεργασία η οποία εφαρμόζεται σε επίπεδο pixels, αλλά είναι πιο απαιτητική από το lighting καθώς χρειάζεται συνδυασμός τόσο του source όσο και του destination pixel
- Δίνουμε παρακάτω τον ορισμό μίας σχετικά απλής παραλλαγής της γνωστής blit την οποία ονομάζουμε trans blit
 - ένα source pixel P_s γίνεται rendered στο αντίστοιχο destination pixel P_d με transparency value $T \in [0,256]$ βάσει της παρακάτω λογικής:
 - Εάν $T = 0$ (δηλ. αδιαφανές), τότε $P_d = P_s$
 - Εάν $T = 255$ (δηλ. διαφανές), τότε P_d δεν αλλάζει
 - Άλλιώς, $P_d = P_d + (P_s - P_d) \times (1 - T / 255)$

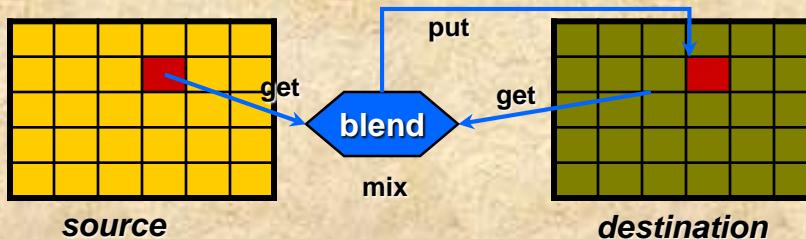
Η λογική
είναι ίδια με
το alpha
blending

Transparency (2/6)

- Το βασικό blending function, η αλλιώς και blender function, είναι αυτό που φαίνεται παρακάτω

```
typedef unsigned char trans_t;  
  
RGBValue transparent (RGBValue src, RGBValue dest, trans_t t)  
{ return dest + (src - dest) * (1 - t / 255.0); }
```

- Η υλοποίηση της trans blit είναι προφανής καθώς πρέπει να εφαρμόζει απλώς μία επιπλέον get pixel για το destination bitmap για να λάβει το destination pixel value.





Transparency (3/6)

- Καθώς αρχίζουμε να σκεφτόμαστε την περίπτωση pre-caching αντιμετωπίζουμε το γεγονός ότι θέλουμε πίνακα από 256x256x256 bytes
 - για indexing **[trans][src][dest]**
 - δηλ. 16 MB, το οποίο μάλλον δεν πρέπει να μας ενθουσιάζει ιδιαίτερα
- Εδώ ρωτάμε «χρειαζόμαστε όλα τα επίπεδα διαφάνειας από 0...255 στο παιχνίδι»
 - Η πιο συνηθισμένη απάντηση είναι «όχι»
 - ◆ ενώ σχεδόν πάντα ακολουθεί η επιπλέον παρατήρηση ότι «χρειαζόμαστε μέχρι στιγμής στο παιχνίδι περίπου δύο ή τρία επίπεδα διαφάνειας»
- Έτσι καταλήγουμε να κρίνουμε σκόπιμο να έχουμε για κάθε διαφορετικό αναγκαίο transparency το δικό του πίνακα υπολογισμού
 - Συνήθως στα παιχνίδια είναι ικανοποιητική η χρήση 3 επιπέδων transparency: high (~200) , medium (~130), low (~80).

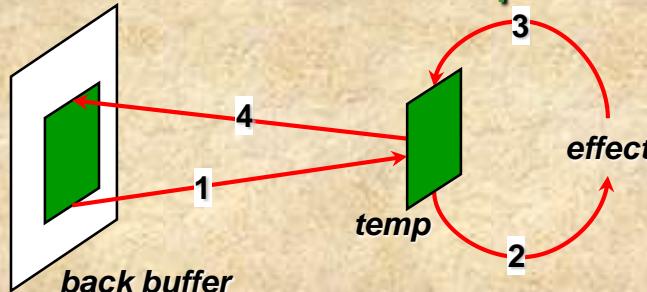


Transparency (4/6)

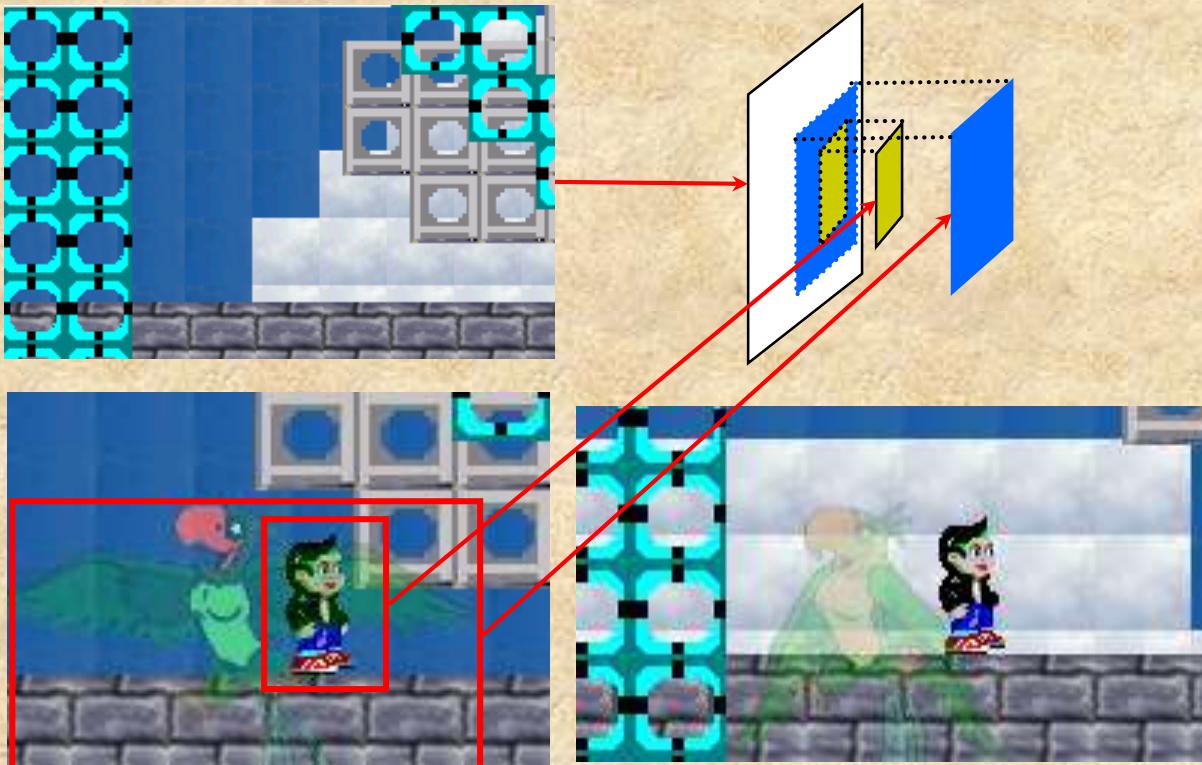
- Ένα πρόβλημα το οποίο δεν είναι αντιληπτό αμέσως είναι ότι τόσο η trans blit όσο και η light / dark blit έχουν πρόσβαση στο destination bitmap
 - Η light / dark blit μόνο για write
 - Η trans blit για read / write
- Αυτό σημαίνει ότι εάν έχετε τέτοιου είδους effects πάνω στο βασικό terrain, το να επιχειρήσετε effect blit απευθείας στον back buffer δεν είναι πολύ σοφό
 - ειδικά εάν πρόκειται για video bitmap, για τα οποία ξέρουμε ότι η άμεση πρόσβαση στην pixel memory κοστίζει δυσάρεστα σε χρόνο

Transparency (5/6)

- Η τακτική που ακολουθούμε σε αυτή την περίπτωση είναι η εξής:
 - χρησιμοποιούμε ένα βοηθητικό **memory bitmap** **temp** με διστάσεις ακριβώς ίδιες με το blit rectangle
 - κάνουμε **fast blit** από **back buffer** → **temp**
 - εφαρμόζουμε το **effect** → **temp**
 - κάνουμε **fast blit** από **temp** → **back buffer**



Transparency (6/6)



1. Render all back layers and action layers
2. Render the sprite
3. Fast blit parrot area (blue) over temp bitmap
4. Trans blit parrot over temp bitmap
5. Fast blit temp over parrot area

Ένα σενάριο για *real-time transparent blit*



Περιεχόμενα

- Τι είναι ειδικά effects
- Lighting / darkening
- Transparency
- ***Scaling***
- Rotation

Scaling (1/5)

- Φαινομενικά μία απλή περίπτωση επεξεργασίας, που ίσως να αναρωτιέται κάποιος γιατί θεωρείται ειδικό effect
 - ↳ γιατί πρέπει να λαμβάνει χώρα σε πραγματικό χρόνο χωρίς *pre-rendering*
 - μπορεί να έχει δύο βασικούς τύπους
 - ◆ *shrink* 
 - ◆ *grow* 
 - αλλά σίγουρα δεν θα αποφύγετε τον πειρασμό να θεωρήσετε ως γενική περίπτωση το stretching



Scaling (2/5)

- Τα shrink και grow χρειάζονται μόνο μία παράμετρο διατηρώντας τις αναλογίες των πλευρών
- Το stretching δίνει δυνατότητα για οποιαδήποτε είδους αλλαγή διαστάσεων
- Ωστόσο σπάνια θα χρειαστεί στο παιχνίδι να εφαρμοστεί stretching
 - γενικά «χαλάει» η αρχική αισθητική των χαρακτήρων
- Αλλά είναι πιθανό να χρειαστείτε shrink / grow effects
 - εάν κάτι τέτοιο δεν είναι απαραίτητο να εφαρμοστεί αναλογικά αλλά υπάρχουν συγκεκριμένοι και λίγοι shrink / grow factors που πρέπει να εφαρμοστούν σε συγκεκριμένα films
 - ◆ ...τότε ξεχνάμε το effect και επιστρέφουμε στο pre-rendering (περισσότερα data αλλά πολύ γρηγορότερο παιχνίδι)

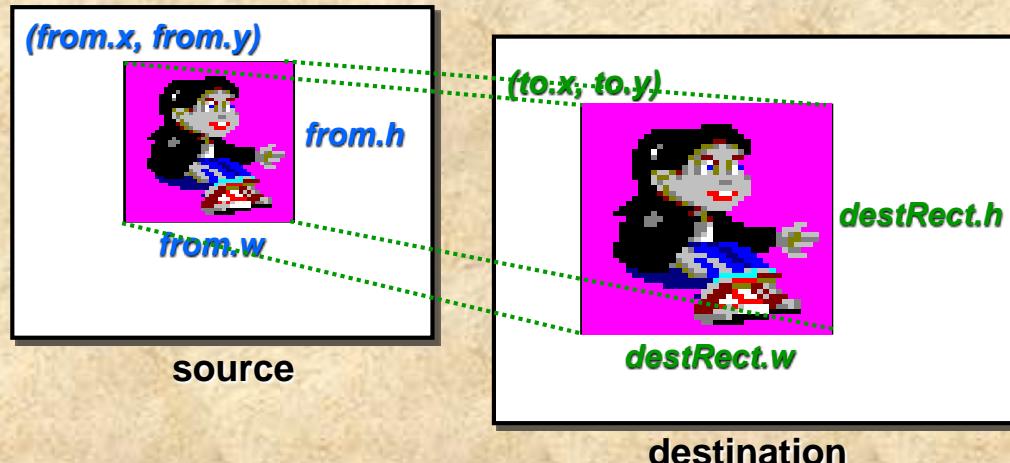


Scaling (3/5)

- To scaling εφαρμόζει τη δημιουργία του destination bitmap με δειγματοληψία από το source bitmap
 - η βασική ανακύκλωση γίνεται πάντα ως προς τις διαστάσεις του destination
 - καθώς πρόκειται για pixel perfect αλγόριθμο πρέπει να αποφύγετε μεγάλα destination bitmaps (συνάρτηση του grow factor και των διαστάσεων του source bitmap)
 - όταν ξεφύγουμε αρκετά από το αρχικό bitmap έχουμε προβλήματα ποιότητας (τόσο στο shrinking όσο και στο grow)
 - στην περίπτωση του grow πρέπει να ελέγχουμε εάν λόγω του scaling φεύγουμε εκτός των ορίων
 - καλό είναι να ορίσετε κάποιο όριο για το shrink και grow, π.χ. min 0.3 για shrink και 2 για grow

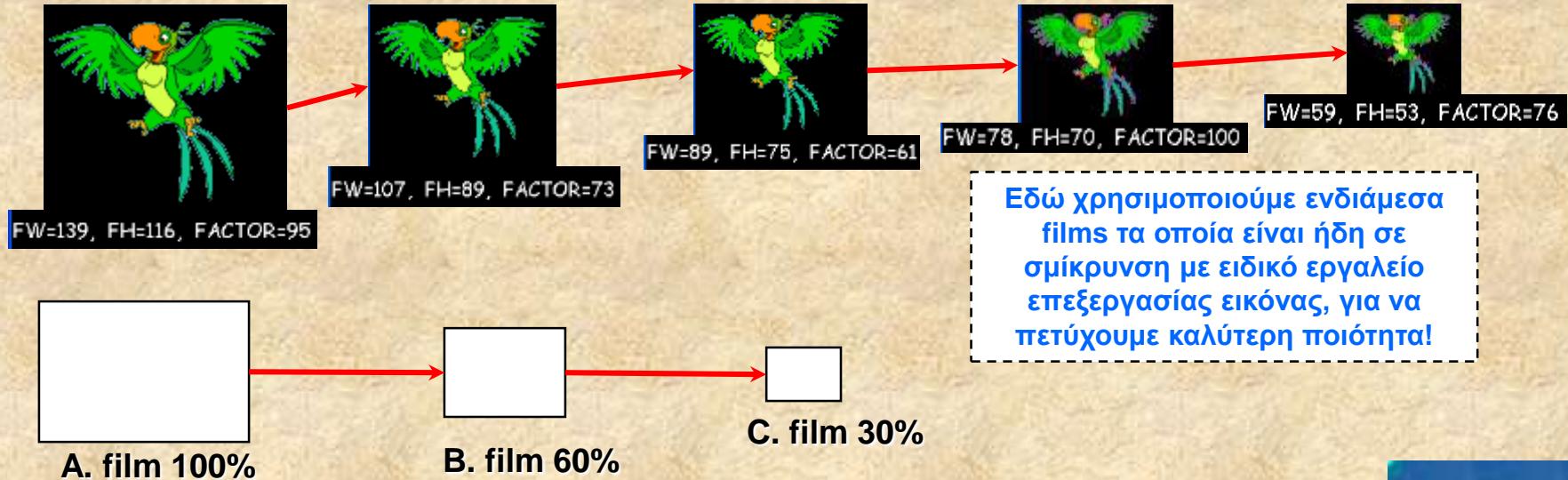
Scaling (4/5)

```
ScaleBlit (Bitmap src, const Rect& from, Bitmap dest, const Point& to, float factor) {  
    Rect destRect(0, 0, from.w * factor, from.h * factor);  
    for each x : 0 to destRect.w-1 do  
        for each y : 0 to destRect.h-1 do {  
            Let C = GetPixel(src, from.x + x / factor, from.y + y / factor);  
            PutPixel(dest, to.x + x, to.y + y, C);  
        }  
    }  
}
```





Scaling (5/5)



Εδώ χρησιμοποιούμε ενδιάμεσα films τα οποία είναι ήδη σε σμίκρυνση με ειδικό εργαλείο επεξεργασίας εικόνας, για να πετύχουμε καλύτερη ποιότητα!



Τα πράγματα δυσκολεύουν όταν επιθυμούμε τα sprites να είναι και scalable αλλά και λειτουργικά (απαιτείται scaling και των collision masks ή bounding areas)



Μήπως ξεχάσαμε κάτι για το scaling (1/2)

- Υπάρχουν οι ανεπιθύμητες διαιρέσεις (slow) και βέβαια θέλουμε δειγματοληψία με τη μεγαλύτερη δυνατή ταχύτητα
- Ας ασχοληθούμε πρώτα με το shrinking και ας θεωρήσουμε shrink factors $f \in (0, 100]$, δηλ. $f/100$ είναι ο πραγματικός παράγοντας για scaling
- Αυτό σημαίνει ότι εάν έχω ένα upper limit για τις διαστάσεις των shrinking sprites, π.χ. 512 pixels (που είναι πολύ καλό πάνω όριο),
 - μπορώ να έχω έναν πίνακα `unsigned short shrinkCoords[100][512];`
 - όπου για κάθε διάσταση d άξονα X ή Y, με to d να αφορά πάντα το destination bitmap
 - το στοιχείο `shrinkCoords[f][d]` είναι η αυθεντική non-scaled τιμή στο source bitmap



Μήπως ξεχάσαμε κάτι για το scaling (2/2)

```
#define ILLEGAL_COORD 0xffff
#define MAX_BITMAP_COORD 512
#define MAX_SCALE_FACTOR 100
#define MIN_SCALE_FACTOR 10

Dim shrinkCoords[MAX_SCALE_FACTOR][MAX_BITMAP_COORD];

void ProduceShrinkCoords (void) {
    for (byte factor = MIN_SCALE_FACTOR; factor <= MAX_SCALE_FACTOR; ++factor)
        for (byte coord = 0; coord < MAX_BITMAP_COORD; ++coord) {
            shrinkCoords[factor][coord] = (100.0 / factor) * coord;
            if (shrinkCoords[factor][coord] >= MAX_BITMAP_COORD)
                shrinkCoords[factor][coord] = ILLEGAL_COORD;
        }
}
```

Προφανώς και δεν περιμένετε να μείνει έτσι αυτή η συνάρτηση με get / put pixel, αλλά να υλοποιηθεί με άμεση χρήση του pixel memory

```
ScaleBlit (Bitmap src, const Rect& from, Bitmap dest, const Point& to, byte factor) {
    Rect destRect(0, 0, from.w * factor/100.0, from.h * factor/100.0);
    auto* pt = shrinkCoords[factor];
    for each x : 0 to destRect.w-1 do
        for each y : 0 to destRect.h-1 do {
            Let C = GetPixel(src, from.x + pt[x], from.y + pt[y]);
            PutPixel(dest, to.x + x, to.y + y, C);
        }
}
```



Περιεχόμενα

- Τι είναι ειδικά effects
- Lighting / darkening
- Transparency
- Scaling
- ***Rotation***

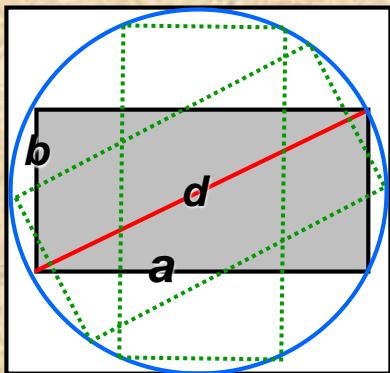


Rotation (1/6)

- Η δυσκολότερη περίπτωση για όλους τους παρακάτω λόγους:
 - η γεωμετρία της είναι πιο πολύπλοκη αλγορίθμικά
 - περισσότερο απαιτητική σε ταχύτητα γιατί εμπλέκει συναρτήσεις ημίτονου και συνημίτονου
 - το destination bitmap δημιουργείται με δειγματοληψία ως προς το source bitmap
 - ◆ αλλά η δειγματοληψία γίνεται σε ανεξάρτητες συντεταγμένες του source (όχι π.χ. όλα τα pixels μίας γραμμής)
 - το destination bitmap δεν έχει απαραίτητα τις ίδιες διαστάσεις με το source bitmap
 - εάν θέλετε να κάνετε περιστρεφόμενα sprites τα οποία εμπλέκονται και σε collision θα πρέπει να φροντίσετε να περιστρέφετε και τα collision masks



Rotation (2/6)



Άρα με source bitmap (a, b) θέλω $\sqrt{a^2+b^2}$ ως διάσταση του destination rotation bitmap (square).

- Η περιστροφή του source bitmap απαιτεί τόσο χώρο όσο ο κύκλος με διáμετρο τη διαγώνιο του source bitmap.
- Αυτό σημαίνει ότι το destination bitmap πρέπει να είναι ένα τετράγωνο στο οποίο μπορεί να εγγραφεί αυτός ο κύκλος.

- Η περιστροφή γίνεται πάντα ως προς το κέντρο και με κάποια γωνία θ .
- Για κάθε rotation χρησιμοποιούμε πάντα το source bitmap.
- Δηλ. εάν θέλουμε δύο διαδοχικά rotations θ_1 και θ_2 , εφαρμόζουμε μία περιστροφή $(\theta_1+\theta_2)$ στο source και όχι θ_1 στο source και θ_2 στο αποτέλεσμα της 1^{ης} περιστροφής.

Έστω το destination bitmap με διαστάσεις $d \times d$. Τότε η δειγματοληψία από το source bitmap γίνεται ως εξής (η αρχή των αξόνων $(0,0)$ τοποθετείται στο κέντρο του bitmap):

$$\begin{aligned} \forall x \in [-d/2, d/2] \quad \forall y \in [-d/2, d/2]: \\ x_{\text{source}} &= \cos(\theta) * x + \sin(\theta) * y + a/2 \\ y_{\text{source}} &= \cos(\theta) * y - \sin(\theta) * x + b/2 \end{aligned}$$

Εάν $x_{\text{source}} \notin [0, a] \vee y_{\text{source}} \notin [0, b]$ το σημείο αγνοείται (clipped).



Rotation (3/6)

Επιλέγουμε τη γωνία *theta* με τύπο *byte* να έχει μέγιστη ακρίβεια δύο μοιρών, έχοντας τιμή *γωνία / 2*. Δηλ. η πραγματική γωνία υπολογίζεται ως *theta << 2*. Στον παρακάτω αλγόριθμο ας θεωρήσουμε ότι: (a) πάντα έχουμε θετική γωνία, (b) πάντα $\in [0, 360)$, (c) οι *cos* και *sin* δέχονται όρισμα σε μοίρες, (d) το *dest* από το σημείο *to* έχει ήδη τις απαιτούμενες για περιστροφή διαστάσεις.

```
RotateBlit (Bitmap src, const Rect& from, Bitmap dest, const Point& to, byte theta) {
    theta<<=2;
    Dim d = sqrt(sqr(from.w)+sqr(from.h));
    for each x : -d/2 to d/2-1 do
        for each y : -d/2 to d/2-1 do {
            Dim xs = cos(theta)*x + sin(theta) * y + from.w/2;
            Dim ys = cos(theta)*y - sin(theta) * x + from.h/2; δειγματοληψία
            Color c;
            if (xs > 0 && xs < from.w && out of bounds test (clipping)
                ys > 0 && ys < from.h &&
                (c = GetPixel(src, from.x + xs, from.y + ys) != GetColorKey()) transparency check
                PutPixel(dest, to.x + (x+d/2), to.y + (y+d/2), c);
        }
    }
```



Rotation (4/6)

- Το πιο προφανές και αναγκαίο optimization στον προηγούμενο κώδικα είναι η αποφυγή υπολογισμού των χρονοβόρων τριγωνομετρικών συναρτήσεων cos και sin
- Αυτό γίνεται με pre-calculated πίνακες εύκολα καθώς έχουμε διακριτές τιμές στη γωνία.

```
#define MAX_ANGLE 360
#define PI          3.1415926535
#define torads(a)   ((a)/360.0 * 2 * PI)

static float sinTable[MAX_ANGLE];
static float cosTable[MAX_ANGLE];
void ProduceTrigTables (void) {
    for (unsigned short i=0; i<MAX_ANGLE; ++i) {
        sinTable[i] = sin(torads(i));
        cosTable[i] = cos(torads(i));
    }
}
```



Rotation (5/6)

- Θα πετύχετε εάν πολύ σημαντικό optimization με αυτό τον τρόπο αλλά και πάλι θα έχετε προβλήματα ταχύτητας. Πιο προχωρημένες επιπλέον βελτιώσεις είναι κυρίως οι παρακάτω:
 - τα **from.w/2**, **from.h/2** και **d/2** υπολογίζονται εκτός του loop σε local variables (και φυσικά με `>>`)
 - η put pixel στο destination πρέπει να αντικατασταθεί με άμεση πρόσβαση σε pixel memory μέσω address που απλώς γίνεται offset στο loop
 - οι αργοί πολλαπλασιασμοί **cosTable[theta]*x**, κλπ, ειδικά λόγω του ότι έχουμε float values, πρέπει να γίνουν pre cached για κάθε theta και **λογικό range x** (π.χ. έως 256) – αυτό αυξάνει σημαντικά την ταχύτητα



Rotation (6/6)

- Το rotation γίνεται πιο απαιτητικό εάν πρέπει να συνδυαστεί με sprites
 - και ειδικότερα εάν πρέπει να γίνονται
 - ◆ έλεγχοι για collision, γεγονός που σημαίνει ότι απαιτείται και περιστροφή των collision masks
 - ◆ animation, γεγονός που σημαίνει ότι σε κάθε frame change χρειάζεται να εφαρμόζεται η περιστροφή
 - ◆ υπολογισμός του minimal bound box
 - όμως ορισμένα από αυτά **υπορούν να γίνουν pre-cached** για την μέγιστη δυνατή ακρίβεια και ταχύτητα
 - ◆ ειδικά ο υπολογισμός του *minimal bound box* μετά την περιστροφή, για κάθε γωνία!





Rotation - επίλογος

```
util_i16* sinMulY = sinMul + firstIndex; \
util_i16* cosMulY = cosMul + firstIndex; \
\
do { \
    util_ui16 x = a - 1; \
    util_i16* sinMulX = sinMul + firstIndex; \
    util_i16* cosMulX = cosMul + firstIndex; \
    \
    do { \
        \
        util_ui16 xp = *cosMulX + *sinMulY + half_sw; \
        util_ui16 yp = *cosMulY - *sinMulX + half_sh; \
        \
        if (xp < sw && yp < sh) { \
            \
            ALADIN_Color srcColor = ALADIN_PixelRead##bits(srcAddr + (yp * srcPitch) + mulbytes(xp)); \
            \
            if (srcColor != transColor) { \
                ALADIN_PixelWrite##bits(destLine, srcColor); \
                MASK_SETBIT; \
            } \
            \
            addbytes(destLine); \
            ++sinMulX; \
            ++cosMulX; \
            MASK_NEXTBITS; \
        } \
        \
        destLine += destPitch; \
        ++sinMulY; \
        ++cosMulY; \
    } \
} while (y--); \
}
```

Απόσπασμα από optimized rotation template (macro), παλιά έκδοση, ALADIN library (Savidis, 2002)

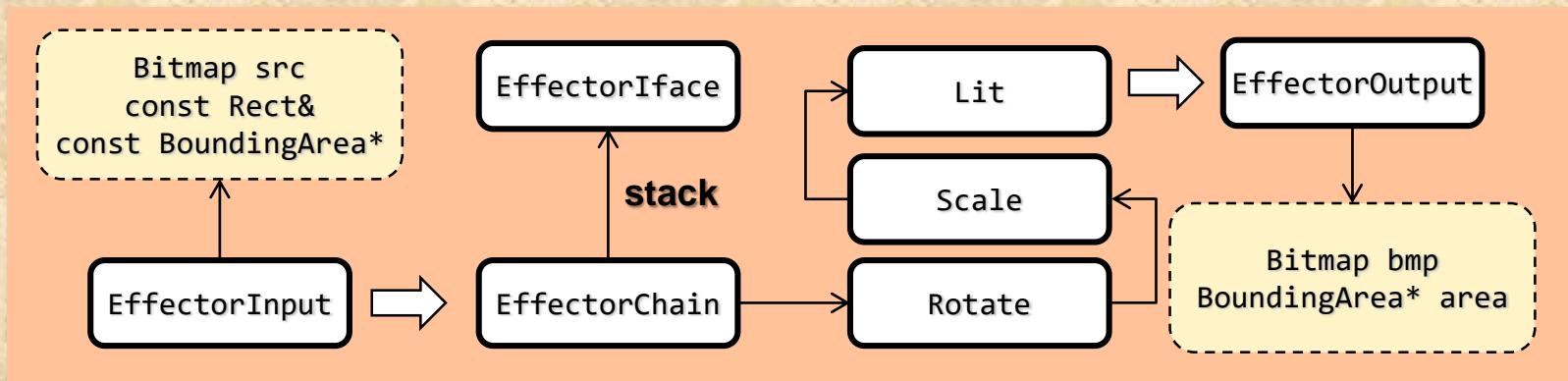
$x \cdot \cos(\theta) + y \cdot \sin(\theta) + w/2$
 $y \cdot \cos(\theta) + x \cdot \sin(\theta) + h/2$

Μετά από πολλά optimizations ο αυθεντικός αλγόριθμος «χάνεται» και καλό είναι να τον έχετε τεκμηριώσει με πολύ ακρίβεια, καθώς και τα ίδια τα optimizations που εφαρμόζετε



Ένθετο για τα effects (1/2)

- Η υλοποίηση των effects που εφαρμόζονται μόνο στο rendering (illumination, transparency) δεν επηρεάζει τις διαστάσεις, ενώ το rotation και scaling τα επηρεάζει
- Ωστόσο, επειδή θέλουμε να συνδυάζονται ελεύθερα καλό είναι να υλοποιηθούν μέσω ενώ κοινού interface το οποίο προαιρετικά μπορεί να έχει ως output το νέο bounding area
- Κάθε effector κάνει cache το ενδιάμεσο αποτέλεσμα το οποίο παρέχει στον επόμενο στη λίστα





Ένθετο για τα effects (2/2)

- Κάτι που δεν γίνεται άμεσα αντιληπτό είναι ότι τα effects lighting / darkening / transparency εμπεριέχουν υπολογισμό κατά την εκτύπωση (display-time calculations)
- ...ενώ τα effects scaling και rotation κατά την «εφαρμογή» (όταν δηλ. αλλάξει στο scale factor ή rotation angle)
- Επειδή κάτι τέτοιο θα αλλάξει την «ισορροπία» στο game engine, καθώς μπορεί μέσα από το game logic να έχουμε αλλαγές μεγέθους και περιστροφές, πρέπει και τα δύο τελευταία να εφαρμόζονται on-rendering μόνο (δηλ. relaxed)
 - και μόνο την πρώτη φορά του rendering σε εσωτερικά βιοηθητικά bitmaps των sprites
- Μία άλλη μεγάλη διαφορά είναι ότι:
 - εάν ένα rotated sprite γίνει displayed σε πέντε σημεία το κόστος είναι: **1 rotation, 5 blits**
 - εάν ένα light / dark / trans sprite γίνει displayed σε πέντε σημεία το κόστος είναι: **5 light / dark / trans blits**