

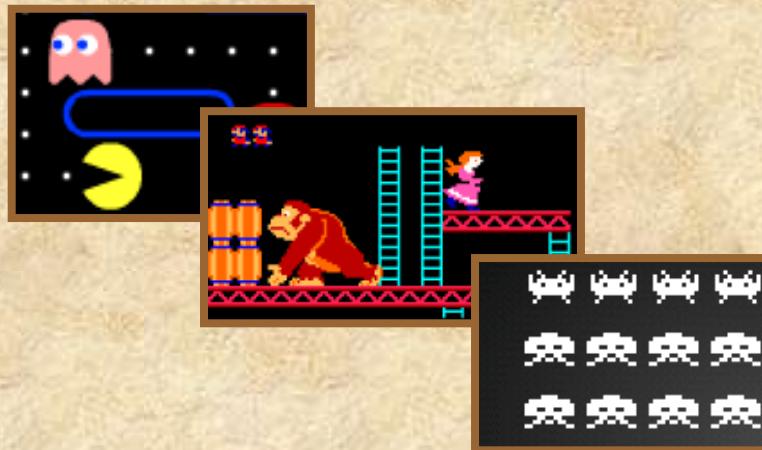


**HY454 : ΑΝΑΠΤΥΞΗ ΕΞΥΠΝΩΝ ΔΙΕΠΑΦΩΝ ΚΑΙ
ΠΑΙΧΝΙΔΙΩΝ**

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



**ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης**



ΑΝΑΠΤΥΞΗ ΠΑΙΧΝΙΔΙΩΝ, Διάλεξη 8η

Animation types and animators



Περιεχόμενα

- ***Animations***
- Animators
- Special details
- Timing



Animations (1/15)

- Animation είναι μία δομή δεδομένων της οποίας στιγμιότυπα μπορούν να περιγράφουν πλήρως μία εκάστοτε κίνηση ενός χαρακτήρα
 - ως προς το είδος της κίνησης,
 - και ως προς το χρονισμό της κίνησης
- Τα animations μπορούν γενικότερα να θεωρηθούν ως γεγονότα που δρομολογούνται με προκαθορισμένο χρονισμό τα οποία ανάλογα με το είδος της ενέργειας που προκαλούν μπορεί:
 - να κινούν χαρακτήρες του παιχνιδιού
 - να προκαλούν scrolling κάποιου layer
 - να εμφανίζουν / κρύβουν αντικείμενα
 - να εκτελούν κάποια ενέργεια που επηρεάζει τα animations



Animations (2/15)

- Ορισμένες βασικές κατηγορίες animations είναι οι παρακάτω:
 - **Moving**, απλή μετακίνηση, λέγεται και *displacement*
 - **Frame range**, αλληλουχία διαδοχικών frames ενός animation film, λέγεται και *sequence*
 - **Frame list**, αλληλουχία συγκεκριμένων οποιονδήποτε frames ενός animation film, λέγεται και *montage*
 - **Moving path**, αλληλουχία από διαφορετικές μετακινήσεις με ταυτόχρονη (προαιρετική) εναλλαγή frame
 - **Flashing**, εναλλαγή κατάστασης για το visibility state ενός χαρακτήρα
 - **Scrolling list**, αλληλουχία εντολών scrolling για κάποιο layer (action ή horizon)



Animations (3/15)

- στα games διαχειρίζομαστε πολλά animation instances
- η χρήση μεταβλητών για κάθε ένα από αυτά ξεχωριστά είναι ασύμφορη (όπως και για τα animation films και sprites)
- σχεδιάζουμε τη διαχείριση των animations να επιτρέπει γρήγορη πρόσβαση σε οποιοδήποτε animation instance βάσει κάποιου id

```
class Animation {  
protected:  
    std::string id;  
public:  
    const std::string& GetId (void) { return id; }  
    void SetId (const std::string& _id);  
    virtual Animation* Clone (void) const = 0;  
    Animation (const std::string& _id) : id(_id){}  
    virtual ~Animation(){}
};
```

Animations (4/15)

■ *Moving animation (1/2)*

- Ορίζει μία μετακίνηση με συγκεκριμένο (**dx,dy**), καθώς και **delay** μετά την πάροδο του οποίου η κίνηση θα εφαρμοστεί.
- Ενώ μπορεί επίσης να οριστεί εάν η μετακίνηση αυτή πρέπει να επαναλαμβάνεται συνεχώς (**continuous**)
 - ◆ |**delay|move|delay|move|...**
- Αυτού του είδους το animation μπορεί να εφαρμοστεί σε όλων των ειδών τις κινήσεις που δεν έχουν εναλλαγή frames, π.χ.
 - ◆ Διαστημόπλοια $\leftarrow \rightarrow$ με input control, σφαίρες, βόμβες, αντικείμενα
- **Σχεδόν πάντα** τέτοιου είδους animations μεταβάλλονται δυναμικά από το πρόγραμμα αλλάζοντας το delay ή / και το dx, dy ανάλογα με τους κανόνες φυσικής της κίνησης
 - ◆ *Προσοχή στις εξισώσεις U,S,t, γ, κλπ.*





Animations (5/15)

■ Moving animation (2/2)

```
class MovingAnimation : public Animation {
protected:
    unsigned          reps = 1; // 0=forever
    int               dx = 0, dy = 0;
    unsigned          delay = 0;
public:
    using Me = MovingAnimation;
    int              GetDx (void) const           { return dx; }
    Me&             SetDx (int v)                { dx = v; return *this; }
    int              GetDy (void) const           { return dy; }
    Me&             SetDy (int v)                { dy = v; return *this; }
    unsigned         GetDelay (void) const        { return delay; }
    Me&             SetDelay (unsigned v)         { delay = v; return *this; }
    unsigned         GetReps (void) const         { return reps; }
    Me&             SetReps (unsigned n)          { reps = n; return *this; }
    bool             IsForever (void) const       { return !reps; }
    Me&             SetForever (void)            { reps = 0; return *this; }
    Animation*      Clone (void) const override { return new MovingAnimation(id, reps, dx, dy, delay); }
MovingAnimation (
    const std::string& _id, unsigned _reps, int _dx, int _dy, unsigned _delay
): Animation(_id), reps (_reps), dx(_dx), dy(_dy), delay(_delay) {}
};
```

Animations (6/15)

- *Frame range (1/2)*
 - Ορίζει μία κίνηση η οποία κυρίως προσδιορίζεται από την αλληλουχία frames, ειδικότερα μεταξύ ενός αρχικού και ενός τελικού frame
 - ◆ Ταυτόχρονα μπορεί να εφαρμόζεται και μετακίνηση με κάθε εναλλαγή frame, η οποία όμως πρέπει να είναι η ίδια για κάθε αλλαγή frame
- Πέρα από την χρήση του για κινήσεις βασικών χαρακτήρων, μπορεί να χρησιμοποιηθεί και για διακοσμητικούς, χωρίς να είναι αναγκαίο να χρησιμοποιηθούν πολλά frames, π.χ.
 - ◆ Φύση, «τρεχούμενο νερό», κινούμενα κλαδιά, μέλισσες / πεταλούδες, ...
 - ◆ Φωτεινές πινακίδες, τηλεοράσεις που υποτίθεται ότι Ιείναι σε λειτουργία, καπνός σε εξατμίσεις αυτοκινήτων, είτε κινούνται είτε όχι





Animations (7/15)

■ Frame range (2/2)

```
class FrameRangeAnimation : public MovingAnimation {  
protected:  
    unsigned          start = 0, end = 0;  
public:  
    using Me = FrameRangeAnimation;  
    unsigned          GetStartFrame (void) const  
    Me&              SetStartFrame (unsigned v)  
    unsigned          GetEndFrame (void) const  
    Me&              SetEndFrame (unsigned v)  
    Animation*       Clone (void) const override {  
        return new FrameRangeAnimation(  
            id, start, end, GetReps(), GetDx(), GetDy(), GetDelay()  
        );  
    }  
    FrameRangeAnimation (  
        const std::string& _id,  
        unsigned s, unsigned e,  
        unsigned r, int dx, int dy, int d  
    ): start(s), end(e), MovingAnimation(id, r, dx, dy, d){}  
};
```

Ο τρόπος εφαρμογής του animation αυτού πρέπει να είναι:

$f_1, delay, f_2, delay, \dots, delay, f_n$, με ταυτόχρονη εφαρμογή (dx, dy) για κάθε f_i

Animations (8/15)

■ Frame list (1/2)

- Πρόκειται για επιλογή οποιονδήποτε frames από ένα animation film τα οποία θα αποτυπωθούν με τη σειρά που εμφανίζονται σε αυτή τη λίστα (και όχι στο ίδιο το film)
 - ◆ Οι λόγοι για τους οποίους χρειάζεται τέτοιου είδους animation μπορεί να είναι είτε η βελτιστοποιημένη αποθήκευση σε compacted films που δεν επιτρέπει τη χρήση frame range, η / και
 - ◆ το είδος της κινηματικής να δίνει τη δυνατότητα διαφορετικών συνδυασμών των frames για μεγαλύτερο πλουραλισμό κινήσεων



0	1	2	3	4	5	6	5	1	0
---	---	---	---	---	---	---	---	---	---

frame list για δύο πλήρη βήματα και επαναφορά στην αρχική στάση

0	1	2	3	2	3	4	1
---	---	---	---	---	---	---	---

frame list για δύο διαδοχικές γροθιές



Animations (9/15)

■ Frame list (2/2)

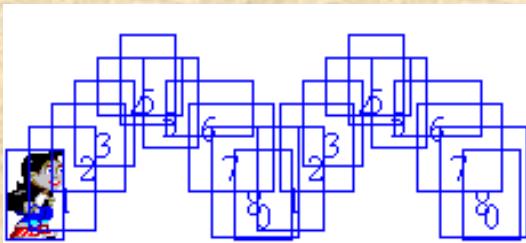
```
class FrameListAnimation : public MovingAnimation {  
public:  
    using Frames = std::vector<unsigned>;  
protected:  
    Frames frames;  
public:  
    const Frames& GetFrames (void) const  
        { return frames; }  
    void SetFrames (const Frames& f)  
        { frames = f; }  
    Animation* Clone (void) const override {  
        return new FrameListAnimation(  
            id, frames, GetReps(), GetDx(), GetDy(), GetDelay()  
        );  
    }  
    FrameListAnimation (  
        const std::string& _id,  
        const Frames& _frames,  
        unsigned r, int dx, int dy, unsigned d, bool c  
    ): frames(_frames), MovingAnimation(id, r, dx, dy, d){}  
};
```

Ο τρόπος εφαρμογής του animation αυτού πρέπει να είναι:
f1, delay,f2, delay,...,delay,fn, με
ταυτόχρονη εφαρμογή (dx,dy) για
κάθε fi

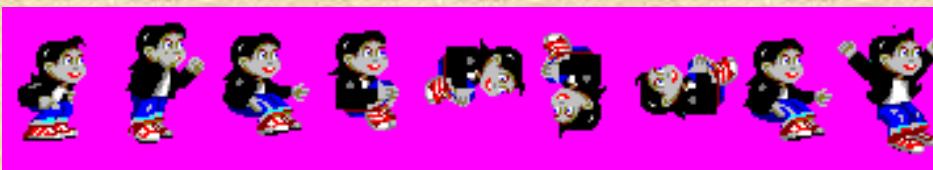
Animations (10/15)

■ Moving path (1/2)

- Πρόκειται για ένα animation που πρακτικά προσφέρει τη μεγαλύτερη ευελιξία για τις περιπτώσεις που απαιτείται διαφοροποίηση των διαδοχικών κινήσεων ως προς τη: μετατόπιση (dx , dy), frame, ακόμη και delay.
- Χρησιμοποιείται σε γενικά περίπλοκες κινήσεις των χαρακτήρων όπως άλματα, ελιγμούς, αναβάσεις, κλπ.



Απαιτεί προσεκτικό σχεδιασμό, με κατάλληλο ορισμό της εφαρμογής των offsets και της εναλλαγής των frames, πάντα θεωρώντας κάποιο συγκεκριμένο frame change alignment





Animations (11/15)

■ Moving path (2/2)

```
struct PathEntry {
    int             dx = 0, dy = 0;
    unsigned        frame = 0;
    unsigned        delay = 0;
    PathEntry (void)=default;
    PathEntry (const PathEntry&)=default;
};

class MovingPathAnimation : public Animation {
public:
    using Path = std::vector<PathEntry>;
private:
    Path path;
public:
    const Path&      GetPath (void) const { return path; }
    void             SetPath (const Path& p) { path = p; }
    Animation*       Clone (void) const override
                      { return new MovingPathAnimation(id, path); }
    MovingPathAnimation (const std::string& _id, const Path& _path) :
        path(_path), Animation(id){}
};
```

Ο τρόπος εφαρμογής του animation αυτού πρέπει να είναι:
 $delay_1, f_1, delay_2, f_2, \dots, delay_n, f_n$, με
ταυτόχρονη εφαρμογή (dx_i, dy_i) για
κάθε f_i



Animations (12/15)

■ *Flashing animation (1/2)*

- Πρόκειται για μία πολύ απλή περίπτωση ουσιαστικά ειδικού effect, το οποίο προκαλεί εναλλακτικά κρύψιμο και εμφάνιση ενός χαρακτήρα για κάποιο χρονικό διάστημα
- Το ενδιαφέρον είναι ότι αυτού του είδους το animation μπορεί να συνδυαστεί με οποιοδήποτε άλλο animation
 - ◆ έτσι ένας χαρακτήρας μπορεί να κάνει flashing ενώ εφαρμόζει οποιοδήποτε άλλο animation
- Συνηθίζεται σε περιπτώσεις που κάποιοι χαρακτήρες υποστούν κάποια «ζημιά» η εξουδετερωθούν, όταν δεν υπάρχει δυνατότητα να δημιουργηθεί ειδικό animation κάτι τέτοιο
 - ◆ Αυτού του είδους τα animations ονομάζονται *punishment feedback animations*



Animations (13/15)

■ *Flashing animation (2/2)*

```
class FlashAnimation : public Animation {  
private:  
    unsigned          repetitions = 0;  
    unsigned          hideDelay = 0;  
    unsigned          showDelay = 0;  
public:  
    using Me = FlashAnimation;  
    Me&               SetRepetitions (unsigned n) { repetitions = n; return *this; }  
    unsigned          GetRepetitions (void) const { return repetitions; }  
    Me&               SetHideDeay (unsigned d)      { hideDelay = d; return *this; }  
    unsigned          GetHideDeay (void) const { return hideDelay; }  
    Me&               SetShowDeay (unsigned d)      { showDelay = d; return *this; }  
    unsigned          GetShowDeay (void) const { return showDelay; }  
    Animation*        Clone (void) const override  
                      { return new FlashAnimation(id, repetitions, hideDelay, showDelay); }  
  
    FlashAnimation (const std::string& _id, unsigned n, unsigned show, unsigned hide) :  
        Animation(id), repetitions(n), hideDelay(hide), showDelay(show){}  
};
```

Ο τρόπος εφαρμογής του animation αυτού πρέπει να είναι:
hideDelay, hide, showDelay, show επαναλαμβανόμενο repetitions φορές.

Animations (14/15)

■ Scrolling list (1/2)

- Όλα τα προηγούμενα animations αφορούσαν sprite instances. Ωστόσο, υπάρχουν περιπτώσεις στις οποίες θέλουμε να έχουμε δρομολογημένες προκαθορισμένες ενέργειες για μετακίνηση του view window σε κάποιο layer
 - ◆ π.χ. για την υλοποίηση ενός effect «σεισμού» θα θέλαμε ένα τρεμούλιασμα σε όλα τα layers, αλλά με διαφορετικό τρόπο στο καθένα.
 - ◆ κάθε τέτοιο τρεμούλιασμα θα πρέπει να οριστεί ως διαδοχικά scrolling commands στο view window με επαναφορά στο αρχικό (όχι απαραίτητα).



Ο συνδυασμός με αντίστοιχα φτηνά animations, όπως πτεραδάκια από πέφτουν από διαφορετικά σημεία, μπορούν να δημιουργήσουν ένα καλό αποτέλεσμα. Αυτά λέγονται amplifying effects.



Animations (15/15)

■ Scrolling list (2/2)

```
struct ScrollEntry {
    int      dx = 0;
    int      dy = 0 ;
    unsigned delay = 0;
};

class ScrollAnimation : public Animation {
public:
    using Scroll = std::vector<ScrollEntry>;
private:
    Scroll scroll;
public:
    const Scroll&      GetScroll (void) const { return scroll; }
    void                SetScroll (const Scroll& p) { scroll = p; }
    Animation*          Clone (void) const override
                        { return new ScrollAnimation(id, scroll); }
    ScrollAnimation (const std::string& _id, const Scroll& _scroll) :
        Animation(_id), scroll(_scroll){}
};
```

Ο τρόπος εφαρμογής του animation αυτού πρέπει να είναι: *delay₁, scroll(dx₁, dy₁), ..., delay_n, scroll(dx_n, dy_n)*



Περιεχόμενα

- Animations
- *Animators*
- Special details
- Timing



Animation management (1/2)

Τα παραλληλόγραμμα παριστάνουν το χρόνο της κάθε ανακύκλωσης του game loop

Κάθε $loop_i$ αρχίζει τη χρονική στιγμή t_i και διαρκεί για διάστημα $t_{i+1} - t_i$

Η μεγάλη γραμμή αυτή παριστάνει το χρόνο κατά τη διάρκεια του παιχνιδιού



Κανόνας Στη μηχανή λαμβάνεται ο παρόντας χρόνος του παιχνιδιού t_i κάθε φορά **κατά την έναρξη του εκάστοτε loop iteration i** με τη συνάρτηση `setgametime()`, ενώ καθόλη τη διάρκεια του loop i και όποτε χρειάζεται η παρούσα ‘τιμή του χρόνου’ χρησιμοποιείται αυτή η τιμή t_i μέσω της `getgametime()`.

```
static unsigned long currTime = 0;  
void setgametime() { currTime = systemtime(); }  
unsigned long getgametime() { return currTime; }
```



Animation management (2/2)

- Έστω ένας animator x ο οποίος δημιουργείται (π.χ. με *new*) και αρχίζει την επεξεργασία (με τη *Start()*) κατά το loop iteration i , δηλαδή τη χρονική t_i .
- Τότε εκ κατασκευής αρχικά το *lastTime* του x είναι το t_i .
- Κατά το loop iteration t_{i+1} θα κληθεί η $x.\text{Progress}(\text{getgametime}())$ το οποίο είναι $x.\text{Progress}(t_{i+1})$.
- Ο animator x θα πρέπει να εφαρμόσει τόσο τμήμα του animation όσο αναλογεί στο χρονικό διάστημα $t_{i+1} - t_i$ και να αναπροσαρμόσει κατάλληλα το *lastTime* ώστε να είναι ίσο με το χρόνο στον οποίο αντιστοιχεί η τελευταία ενάργεια του animation που μόλις εφαρμόστηκε (είναι πάντα $\leq t_{i+1}$).

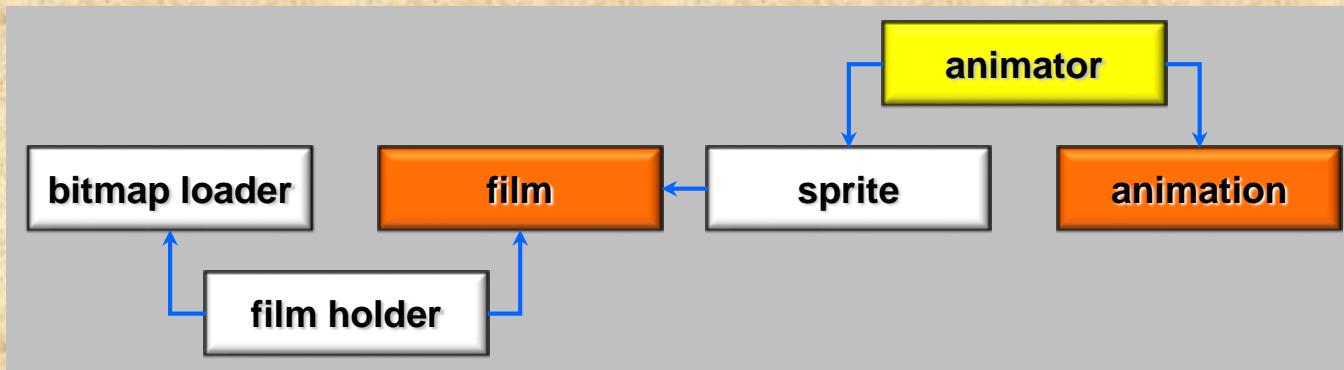


Animators (1/12)

- Τα **animation classes** «δεν γνωρίζουν» πως εφαρμόζονται τα ίδια κατά την εκτέλεση καθώς απλώς φέρουν τα animation data.
- Το **sprite class** όχι απλώς δεν πρέπει να γνωρίζει πως εφαρμόζεται ένα animation, αλλά ούτε καν τι είναι είναι ένα animation
 - Ωστόσο προσφέρει τα αναγκαία methods για να εφαρμόσει κάποιος ένα animation σε ένα sprite instance (δηλ. μετακίνηση και αλλαγή frame)
 - ◆ *Move(dx,dy)*
 - ◆ *SetFrame(frameNo)*
- «Ποιος» λοιπόν ευθύνεται για την εφαρμογή ενός animation?
- ➔ Το **animator class** το οποίο γνωρίζει πώς να εφαρμόσει ένα animation instance σε ένα sprite instance (ή και σε ένα layer)

Animators (2/12)

- Προφανώς χρειάζομαι ειδικό τύπο animator για κάθε διαφορετικό τύπο animation
 - ενώ είναι απαραίτητος ο διαχωρισμός μεταξύ sprite animator και layer animator
 - προφανώς όταν εφαρμόζεται ένα animation instance σε ένα sprite instance πρέπει:
 - ◆ το film instance του sprite να αντιστοιχεί πλήρως σε αυτό το animation instance





Animators (3/12)

Super-class (1/2)

```
typedef uint64_t timestamp_t;
enum animatorstate_t {
    ANIMATOR_FINISHED = 0, ANIMATOR_RUNNING = 1, ANIMATOR_STOPPED = 2
};
class Animator {
public:
    using OnFinish = std::function<void(Animator*)>;
    using OnStart = std::function<void(Animator*)>;
    using OnAction = std::function<void(Animator*, const Animation&)>;
protected:
    timestamp_t lastTime = 0;
    animatorstate_t state = ANIMATOR_FINISHED;
    OnFinish onFinish;
    OnStart onStart;
    OnAction onAction;
    void NotifyStopped (void);
    void NotifyStarted (void);
    void NotifyAction (const Animation&);
    void Finish (bool isForced = false);
public:
    void Stop (void);
    bool HasFinished (void) const { return state != ANIMATOR_RUNNING; }
    virtual void TimeShift (timestamp_t offset);
    virtual void Progress (timestamp_t currTime) = 0;
    template <typename Tfunc> void SetOnFinish (const Tfunc& f) { onFinish = f; }
    template <typename Tfunc> void SetOnStart (const Tfunc& f) { onStart = f; }
    template <typename Tfunc> void SetOnAction (const Tfunc& f) { onAction = f; }
};
```



Animators (4/12)

Super-class (2/2)

```
Animator (void);
Animator (const Animator&) = delete;
Animator (Animator&&) = delete;
virtual ~Animator(){}  
  
void Animator::Finish (bool isForced) {
    if (!HasFinished()) {
        state = isForced ? ANIMATOR_STOPPED : ANIMATOR_FINISHED;
        NotifyStopped();
    }
}
void Animator::Stop (void)
{ Finish(true); }  
  
void Animator::NotifyStopped (void) {
    if (onFinish)
        (onFinish)(this);
}  
  
void Animator::NotifyAction (const Animation& anim) {
    if (onAction)
        (onAction)(this, anim);
}  
  
void Animator::TimeShift (timestamp_t offset)
{ lastTime += offset; }
```

Με την *TimeShift* μπορούμε να κάνουμε pause το game εύκολα, αρκεί να κρατήσουμε το χρόνο για τον οποίο έχει γίνει paused και έπειτα να κάνουμε time shift όλους τους animators (δηλ. τα animator instances).



Animators (5/12)

■ *Derived classes*

- Χρειάζεται ένα για κάθε διαφορετικό τύπο animation.
Δηλ. πρέπει να ορίσουμε:
 - ◆ **MovingAnimator** class
 - ◆ **FrameRangeAnimator** class
 - ◆ **FrameListAnimator** class
 - ◆ **MovingPathAnimator** class
 - ◆ **FlashAnimator** class
- Θα μελετήσουμε την κατασκευή ορισμένων animator classes, καθώς οι υπόλοιπες κλάσεις διαφοροποιούνται μόνο ως προς την λογική εκτέλεσης του animation



Animators (6/12)

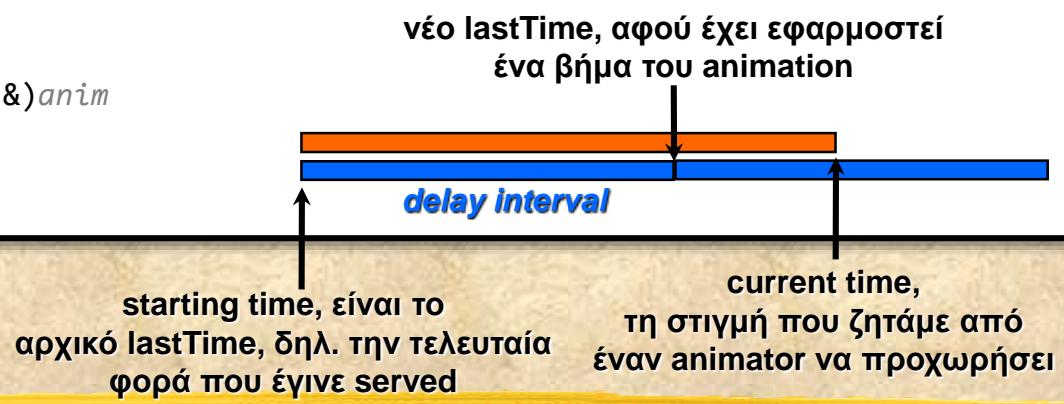
■ *MovingAnimator (1/2)*

```
class MovingAnimator : public Animator {  
protected:  
    MovingAnimation* anim = nullptr;  
    unsigned currRep = 0; // animation state  
public:  
    void Progress (timestamp_t currTime);  
    auto GetAnim (void) const -> const MovingAnimation&  
    { return *anim; }  
    void Start (MovingAnimation* a, timestamp_t t) {  
        anim = a;  
        lastTime = t;  
        state = ANIMATOR_RUNNING;  
        curRep = 0;  
        NotifyStarted();  
    }  
    MovingAnimator (void) = default;  
};
```

Επειδή πρόκειται για πολύ γενικό animation, σχεδιάζουμε την κλάση όσο πιο γενική γίνεται (*no sprite dependency!*)



```
void MovingAnimator::Progress (timestamp_t currTime) {  
    while (currTime > lastTime && (currTime - lastTime) >= anim->GetDelay()) {  
        lastTime += anim->GetDelay();  
        NotifyAction(*anim);  
        if (!anim->IsForever() && ++currRep == anim->GetReps()) {  
            state = ANIMATOR_FINISHED;  
            NotifyStopped();  
            return;  
        }  
    }  
}  
void Sprite_MoveAction (Sprite* sprite, const MovingAnimation& anim) {  
    sprite->Move(anim.GetDx(), anim.GetDy());  
}  
animator->SetOnAction(  
    [sprite](Animator* animator, const Animation& anim) {  
        assert(dynamic_cast<const MovingAnimation*>(&anim));  
        Sprite_MoveAction(  
            sprite,  
            (const MovingAnimation&)anim  
        );  
    }  
);
```





Animators (8/12)

FrameRangeAnimator (1/3)

```
class FrameRangeAnimator : public Animator {  
protected:  
    FrameRangeAnimation* anim = nullptr;  
    unsigned currFrame = 0;           // animation state  
    unsigned currRep = 0;            // animation state  
public:  
    void Progress (timestamp_t currTime);  
    GetCurrFrame (void) const { return currFrame; }  
    GetCurrRep (void) const { return currRep; }  
    Start (FrameRangeAnimation* a, timestamp_t t) {  
        anim      = a;  
        lastTime  = t;  
        state     = ANIMATOR_RUNNING;  
        currFrame = anim->GetStartFrame();  
        currRep   = 0;  
        NotifyStarted();  
        NotifyAction(*anim);  
    }  
    FrameRangeAnimator (void) = default;  
};
```

Υπενθυμίζεται ότι στο frame range animation εφαρμόζεται πάντα το πρώτο frame



Animators (9/12)

■ *FrameRangeAnimator* (2/3)

```
void FrameRangeAnimator::Progress (timestamp_t currTime) {  
  
    while (currTime > lastTime && (currTime - lastTime) >= anim->GetDelay()) {  
  
        if (currFrame == anim->GetEndFrame()) {  
            assert(anim->IsForever() || currRep < anim->GetReps());  
            currFrame = anim->GetStartFrame(); // flip to start  
        }  
        else  
            ++currFrame;  
  
        lastTime += anim->GetDelay();  
        NotifyAction(*anim);  
  
        if (currFrame == anim->GetEndFrame())  
            if (!anim->IsForever() && ++currRep == anim->GetReps()) {  
                state = ANIMATOR_FINISHED;  
                NotifyStopped();  
                return;  
            }  
    }  
}
```



Animators (9/12)

■ *FrameRangeAnimator* (3/3)

```
void FrameRange_Action (Sprite* sprite, Animator* animator, const FrameRangeAnimation& anim) {
    auto* frameRangeAnimator = (FrameRangeAnimator*) animator;
    if (frameRangeAnimator->GetCurrFrame() != anim.GetStartFrame() ||
        frameRangeAnimator->GetCurrRep())
        sprite->Move(anim.GetDx(), anim.GetDy());
    sprite->SetFrame(frameRangeAnimator->GetCurrFrame());
}

animator->SetOnAction(
    [sprite](Animator* animator, const Animation& anim) {
        FrameRange_Action(sprite, animator, (const FrameRangeAnimation&) anim);
    }
);
```



Animators (11/12)

■ Κεντρική διαχείριση των animators (1/3)

- Κάθε instantiated animator είναι registered σε ένα singleton class που λέγεται animator manager
- Αυτό μπορεί εύκολα να υλοποιηθεί στο επίπεδο του super-class καθώς στον constructor μπορεί να γίνεται registered και στον destructor removed
- Εάν όλοι οι animators είναι «μαζεμένοι» σε μία λίστα, μπορώ να κάνω μαζικά σε ένα σημείο progress όλα τα running animations (από τους animators) βάσει του εκάστοτε current time
- Για μεγαλύτερη ταχύτητα έχουμε δύο sets:
 - ◆ **running animators**, δηλ. αυτούς που έχουν ήδη γίνει started,
 - ◆ **suspended animators**, δηλ. αυτούς που έχουν γίνει είτε normally finished, ή explicitly stopped



Animators (12/12)

■ Κεντρική διαχείριση των animators (2/3)

```
class AnimatorManager {  
private:  
    std::set<Animator*> running, suspended;  
    static AnimatorManager singleton;  
    AnimatorManager (void)=default;  
    AnimatorManager (const AnimatorManager&) = delete;  
    AnimatorManager (AnimatorManager&&) = delete;  
public:  
    void Register (Animator* a)  
        { assert(a->HasFinished()); suspended.insert(a); }  
    void Cancel (Animator* a)  
        { assert(a->HasFinished()); suspended.erase(a); }  
    void MarkAsRunning (Animator* a)  
        { assert(!a->HasFinished()); suspended.erase(a); running.insert(a); }  
    void MarkAsSuspended (Animator* a)  
        { assert(a->HasFinished()); running.erase(a); suspended.insert(a); }  
    void Progress (timestamp_t currTime) {  
        auto copied (running);  
        for (auto* a : copied)  
            a->Progress(currTime);  
    }  
    static auto GetSingleton (void) -> AnimatorManager& { return singleton; }  
    static auto GetSingletonConst (void) -> const AnimatorManager& { return singleton; }  
};
```



Animators (12/12)

■ Κεντρική διαχείριση των animators (3/3)

```
void Animator::NotifyStopped (void) {
    AnimatorManager::GetSingleton().MarkAsSuspended(this);
    if (onFinish)
        (onFinish)(this);
}

void Animator::NotifyStarted (void) {
    AnimatorManager::GetSingleton().MarkAsRunning(this);
    if (onStart)
        (onStart)(this);
}

Animator::Animator (void)
{ AnimatorManager::GetSingleton().Register(this); }

Animator::~Animator (void)
{ AnimatorManager::GetSingleton().Cancel(this); }
```

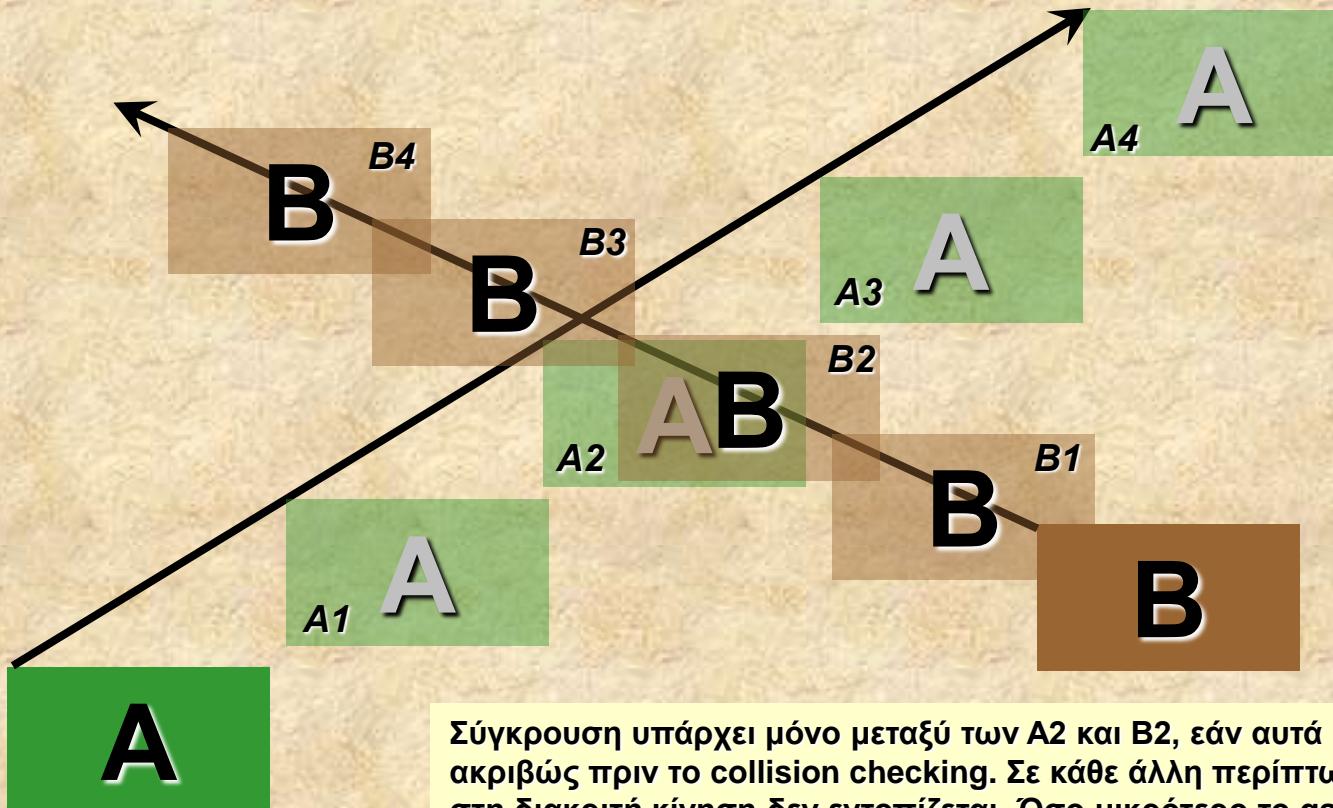


Animation and frame rate (1/2)

- Έχουμε πει ότι δεν μας ενδιαφέρει το παιχνίδι να τρέχει σε production version σε μεγάλα frame rates καθώς στο τέλος υποχρεούμαστε να κάνουμε vsync, δηλ. είμαστε bounded από το display frequency
- Ωστόσο, έαν μπορούμε να τρέχουμε σε μεγάλο game loop rate (π.χ. LPS 200) αλλά να φροντίζουμε να κάνουμε render με το display frequency σημαίνει ότι έχουμε τη δυνατότητα να εφαρμόζουμε animations με μικρό delay (π.χ. 5 mses) και offset (π.χ. 2 pixels)
- Ας δούμε τι πλεονεκτήματα έχει αυτό σε σχέση με ένα χαμηλό loop rate παρουσιάζοντας ένα κλασικό πρόβλημα των διακριτών κινήσεων στον επίπεδο και στον τρισδιάστατο χώρο

Animation and frame rate (2/2)

The tunneling problem



Σύγκρουση υπάρχει μόνο μεταξύ των A2 και B2, εάν αυτά εφαρμοστούν ακριβώς πριν το collision checking. Σε κάθε άλλη περίπτωση το collision στη διακριτή κίνηση δεν εντοπίζεται. Όσο μικρότερο το game loop rate, τόσο μικρότερη και η ακρίβεια του collision detection.



Περιεχόμενα

- Animations
- Animators
- **Secrets**
- Timing



Secrets (1/5)

- Καταστροφή αντικειμένων με «καθυστέρηση / αναβολή»
 - *late / deferred object destruction*
 - Καθώς εκτελείται το game loop, ενδέχεται κάποιο sprite να καταστραφεί λόγω κάποιου «γεγονότος», π.χ. collision
 - Όμως αναφορά σε αυτό το sprite μπορεί να υπάρχει και σε άλλα σημεία, π.χ.
 - ◆ animator progress,
 - ◆ display loop από τον sprite manager,
 - ◆ σε άλλα collision pairs
 - Σε τέτοιες περιπτώσεις, οι τακτικές instant destruction δεν ενδείκνυνται, αλλά σχεδιάζουμε ένα ειδικό software pattern που το ονομάζουμε late / relaxed / lazy destruction pattern



Secrets (2/5)

```
class LatelyDestroyable;
class DestructionManager {
    std::list<LatelyDestroyable*> dead;
    static DestructionManager singleton;
public:
    void Register (LatelyDestroyable* d);
    void Commit (void);
    static auto Get (void) -> DestructionManager& { return singleton; }
};

class LatelyDestroyable {
protected:
    friend class DestructionManager;
    bool alive = true;
    bool dying = false;
    virtual ~LatelyDestroyable() { assert(dying); }
    void Delete (void);
public:
    bool IsAlive (void) const { return alive; }
    void Destroy (void) {
        if (alive) {
            alive = false;
            DestructionManager::Get().Register(this);
        }
    }
    LatelyDestroyable (void) = default;
};
```

Late destruction requires to abandon the `delete` operator (so the destructor must be private in the subclasses too)



Secrets (3/5)

```
void LatelyDestroyable::Delete (void)
{ assert(!dying); dying = true; delete this; }

void DestructionManager::Register (LatelyDestroyable* d) {
    assert(!d->IsAlive());
    dead.push_back(d);
}

void DestructionManager::Commit (void) {
    for (auto* d : dead)
        d->Delete();
    dead.clear();
}

// may adopt this for animators in case we wish to Destroy() in callbacks
// and do not bother to have deleted pointers being used

class Animator : public LatelyDestroyable {
    ...
};
```



Secrets (4/5)

- Ανακύκλωση στιγμιότυπων για κλάσεις με πολύ συχνή δημιουργία και καταστροφή στιγμιότυπων – instance recycling
 - Αυτό είναι το Dynamic Memory Recycler (DMR) pattern, το οποίο περιγράφεται επίσης και στο μάθημα της Τεχνολογίας Λογισμικού (HY352).
 - Εάν σε ένα σύστημα υπάρχει έχει ένα threshold **N** για τον μέγιστο αριθμό των δυναμικών instances μίας κλάσης **C**, για οποιαδήποτε χρονική στιγμή, τότε μόνο με **N** allocations καλύπτω τις ανάγκες του συστήματος σε **C** instances.
 - ◆ Έστω ένα game με average 50 sprite allocations / sec και 100 sprites maximum. Τότε ενώ σε 1 λεπτό θα έπρεπε να κάνω 3000 allocations, αρκούμαι στο μέγιστο αριθμό $N \leq 100$.
 - Εφαρμόζεται η τεχνική αυτή σε συνδυασμό με το late destruction, τόσο για animators όσο και sprites



Secrets (5/5)

Dynamic memory recycler for a class X [adapted for C++11 from original version from I-GET UIMS language, Savidis 1997][more advanced than the CS352 version]

```
template <class T> class Recycled { // turn any class to green (recycle friendly)
protected:
    static std::stack<typename T*> recycler;
    static T* top_and_pop (void)
        { auto* x = recycler.top(); recycler.pop(); return x; }
public:
    template <class ... Types> static T* New (Types ... args) {
        if (recycler.empty())
            return new T (args...); // automatic propagation of any args
        else
            return new (top_and_pop()) T(args...); // reusing ...
    }
    void Delete (void)
        { this->~T(); recycler.push(this); }
};
```

```
class X {
public:
    X (int){}
    X (void){}
    X (const std::string&, int){}
};
```

```
using Recycled_X = Recycled<X>;
auto* x1 = Recycled_X::New();
auto* x2 = Recycled_X::New(1);
auto* x3 = Recycled_X::New("hello", 2);
```

Issue: should cast refs explicitly in `New()`, else they are defined as copied values, thus use `New((const A&) a)`, not merely `New(a)` for `A& a`



Περιεχόμενα

- Animations
- Animators
- Secrets
- *Timing*



Timing (1/5)

- Για το timing στο game engine χρησιμοποιούμε μία κλάση για το system clock καθώς και ένα πολύ ευέλικτο και εκφραστικά ισχυρό νέο ζευγάρι animation-animator

```
#include <chrono>

class SystemClock final {
private:
    std::chrono::high_resolution_clock clock;
    static SystemClock singleton;
public:
    static auto Get (void) -> SystemClock&
    { return singleton; }
    uint64_t milli_secs (void) const;
    uint64_t micro_secs (void) const;
    uint64_t nano_secs (void) const;
};
```



Timing (2/5)

```
uint64_t SystemClock::milli_secs (void) const {
    return std::chrono::duration_cast<std::chrono::milliseconds>
        (clock.now().time_since_epoch()).count();
}

uint64_t SystemClock::micro_secs (void)  const {
    return std::chrono::duration_cast<std::chrono::microseconds>
        (clock.now().time_since_epoch()).count();
}

uint64_t SystemClock::nano_secs(void) const {
    return std::chrono::duration_cast<std::chrono::nanoseconds>
        (clock.now().time_since_epoch()).count();
}

uint64_t GetSystemTime (void) {
    return SystemClock::Get().milli_secs();
}
```



Timing (3/5)

```
class TickAnimation : public Animation {
protected:
    unsigned          delay = 0;
    unsigned          reps = 1;
    bool              isDiscrete = true; // false: when used for custom timed actions
    bool              Inv (void) const { return isDiscrete || reps == 1; }
public:
    using Me = TickAnimation;
    unsigned          GetDelay (void) const           { return delay; }
    Me&               SetDelay (unsigned v)           { delay = v; return *this; }
    unsigned          GetReps (void) const            { return reps; }
    Me&               SetReps (unsigned n)           { reps = n; return *this; }
    bool              IsForever (void) const          { return !reps; }
    Me&               SetForever (void)             { reps = 0; return *this; }
    bool              IsDiscrete (void) const         { return isDiscrete; }
    Animation*        Clone (void) const override     { return new TickAnimation(id, delay, reps, isDiscrete = true); }
    TickAnimation (const std::string& _id, unsigned d, unsigned r, bool discrete) :
        Animation(id), delay(d), reps(r), isDiscrete(discrete) { assert(Inv()); }
};
```



Timing (4/5)

```
class TickAnimator : public Animator {  
protected:  
    TickAnimation*      anim = nullptr;  
    unsigned             currRep = 0;  
    unsigned             elapsedTime = 0; // keep track of time passed between triggers  
  
public:  
    void                Progress (timestamp_t currTime) override;  
    unsigned             GetCurrRep (void) const { return currRep; }  
    unsigned             GetElapsedTime (void) const { return elapsedTime; }  
    unsigned             GetElapsedTimeNormalised (void) const  
        { return float(elapsedTime) / float(anim->GetDelay()); }  
  
    void                Start (const TickAnimation& a, timestamp_t t) {  
        anim           = (TickAnimation*) a.Clone();  
        lastTime       = t;  
        state          = ANIMATOR_RUNNING;  
        currRep        = 0;  
        elapsedTime   = 0;  
        NotifyStarted();  
    }  
  
    TickAnimator (void) = default;  
};
```



Timing (5/5)

```
void TickAnimator::Progress (timestamp_t currTime) {
    if (!anim->IsDiscrete()) { // no discrete fires in every loop!
        elapsedTime = currTime - lastTime;
        lastTime = currTime;
        NotifyAction(*anim);
    }
    else
        while (currTime > lastTime && (currTime - lastTime) >= anim->GetDelay()) {

            lastTime += anim->GetDelay();
            NotifyAction(*anim);

            if (!anim->IsForever() && ++currRep == anim->GetReps()) {
                state = ANIMATOR_FINISHED;
                NotifyStopped();
                return;
            }
        }
}
```