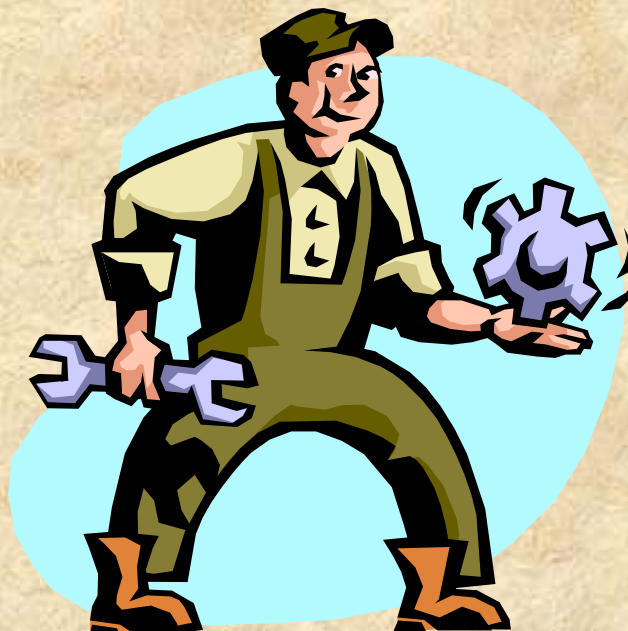


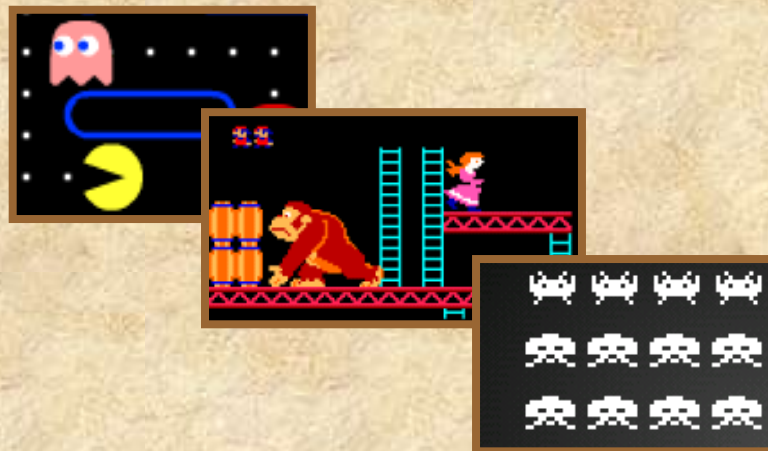


ΗΥ454 : ΑΝΑΠΤΥΞΗ ΕΞΥΠΝΩΝ ΔΙΕΠΑΦΩΝ ΚΑΙ ΠΑΙΧΝΙΔΙΩΝ

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



ΔΙΔΑΣΚΟΝΤΕΣ
Αντώνιος Σαββίδης



**ΑΝΑΠΤΥΞΗ ΠΑΙΧΝΙΔΙΩΝ,
Διάλεξη 9η
Sprites**



Sprites (1/26)

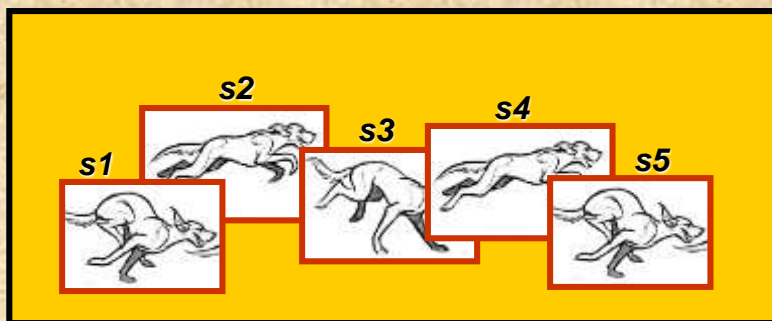
- Basics
- Moving, z-ordering and display
- Frame change
- Gravity
- Collision engagement

Sprites (2/26)

■ Basics (1/2)

- Τα sprites κυρίως έχουν:

- ◆ θέση στο terrain (pixel based)
- ◆ current animation film (pointer)
- ◆ current frame number
- ◆ current bound box (of frame, pointer or copied data)
- ◆ flag για το αν θέλουμε το sprite να είναι visible



Μπορούμε να έχουμε όσα διαφορετικά sprites θέλουμε τα οποία όλα χρησιμοποιούν το ίδιο film, αλλά βρίσκονται σε διαφορετική θέση και frame.



Sprites (3/26)

```
class Clipper;
class Sprite {
public:
    using Mover = std::function<void(const Rect&, int* dx, int* dy)>;
protected:
    byte        frameNo      = 0;
    Rect        frameBox;    // inside the film
    int         x = 0, y = 0;
    bool        isVisible   = false;
    AnimationFilm* currFilm  = nullptr;
    BoundingBox* boundingArea = nullptr;
    unsigned    zorder      = 0;
    std::string typeId, stateId;
    Mover       mover;
    MotionQuantizer quantizer;
public:
    template <typename Tfunc>
    void      SetMover (const Tfunc& f)
        { quantizer.SetMover(mover = f); }

    const Rect GetBox (void) const
        { return { x, y, frameBox.w, frameBox.h }; }

    void      Move (int dx, int dy)
        { quantizer.Move(GetBox(), &dx, &dy); }

    void      SetPos (int _x, int _y) { x = _x; y = _y; }
    void      SetZorder (unsigned z) { zorder = z; }
    unsigned  GetZorder (void) { return zorder; }
```

Διαφορετικές πεταλούδες (sprite instances) σε μία μόνο σκηνή από το παιχνίδι Lomax του ίδιου film.

• Κάθε sprite instance έχει το δικό του ξεχωριστό frame index και frame box, ενώ πουθενά δεν υπάρχει «δικό του» bitmap instance (θυμίζει λίγο το fly weight pattern, αλλά μόνο λίγο...).

• Ο χώρος μνήμης που απαιτεί κάθε sprite είναι πολύ μικρός και θα μπορούσαμε να έχουμε δεκάδες διαφορετικά sprites του ίδιου film σε μία σκηνή χωρίς προβληματισμό για τη μνήμη που απαιτείται.





Sprites (4/26)

- Η κίνηση ενός sprite απλώς σημαίνει την μετατόπιση της θέσης του (x,y) για κάποιο (dx, dy)
 - Η υποστήριξη της κίνησης αφορά κανόνες που εξαρτώνται από την υλοποίηση του terrain
 - ενδεχομένως και να χειρίζονται μέσω third-party physics engine
- Το z-ordering είναι ένας αριθμός που ορίζει την σχετική προτεραιότητα στην αποτύπωση των sprites πάνω στην οθόνη
 - Το sprite με το μικρότερο z-ordering εκτυπώνεται πρώτο ενώ αυτό με το μεγαλύτερο z-ordering εκτυπώνεται τελευταίο
- Το display function των sprites φροντίζει να εκτυπώσει το τμήμα του current frame που είναι ορατό μέσα στα όρια του view window του εκάστοτε terrain
 - καλεί στην πραγματικότητα την display function του animation film, η οποία αλλάζει ελαφρώς για να μπορεί να εφαρμόζει και *clipped blit*



Sprites (5/26)

```
template <typename Tnum>
int number_sign (Tnum x) {
    return x > 0 ? 1 : x < 0 ? -1 : 0;
}

// generic quantizer, can be used to filter motion with any terrain
// motion filtering function

class MotionQuantizer {
public:
    using Mover = std::function<void(const Rect& r, int* dx, int* dy)>;

protected:
    int    horizMax = 0, vertMax = 0;
    Mover  mover; // filters requested motion too!
    bool   used = false;

public:
    MotionQuantizer& SetUsed (bool val);
    MotionQuantizer& SetRange (int h, int v)
        { horizMax = h, vertMax = v; used = true; return *this; }

    MotionQuantizer& SetMover (const Mover & f)
        { mover = f; return *this; }

    void Move (const Rect& r, int* dx, int* dy);

    MotionQuantizer (void) = default;
    MotionQuantizer (const MotionQuantizer&) = default;
};
```




Sprites (6/26)

```
void MotionQuantizer::Move (const Rect& r, int* dx, int* dy) {  
    if (!used)  
        mover(r, dx, dy);  
    else  
        do {  
            auto sign_x = number_sign(*dx);  
            auto sign_y = number_sign(*dy);  
  
            auto dxFinal = sign_x * std::min(horizMax, sign_x * *dx);  
            auto dyFinal = sign_y * std::min(vertMax, sign_y * *dy);  
  
            mover(r, &dxFinal, &dyFinal);  
  
            if (!dxFinal) // X motion denied  
                *dx = 0;  
            else  
                *dx -= dxFinal;  
  
            if (!dyFinal) // Y motion denied  
                *dy = 0;  
            else  
                *dy -= dyFinal;  
        } while (*dx || *dy);  
}
```




Sprites (7/26)

```
void SetFrame (byte i) {
    if (i != frameNo) {
        assert(i < currFilm->GetTotalFrames());
        frameBox = currFilm->GetFrameBox(frameNo = i);
    }
}

byte GetFrame (void) const { return frameNo; }

void SetBoundingArea (const BoundingArea& area)
    { assert(!boundingArea); boundingArea = area.Clone(); }
void SetBoundingArea (BoundingArea* area)
    { assert(!boundingArea); boundingArea = area; }
auto GetBoundingArea (void) const -> const BoundingArea*
    { return boundingArea; }

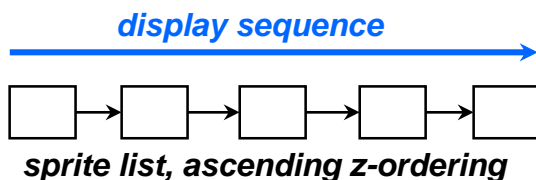
auto GetTypeId (void) -> const std::string& { return typeId; }
void SetVisibility (bool v) { isVisible = v; }
bool IsVisible (void) const { return isVisible; }
bool CollisionCheck (const Sprite* s) const;
void Display (Bitmap dest, const Rect& dpyArea, const Clipper& clipper) const;

Sprite (int _x, int _y, AnimationFilm* film, const std::string& _typeId = ""):
    x(_x), y(_y), currFilm(film), typeId (_typeId)
    { frameNo = currFilm->GetTotalFrames(); SetFrame(0); }
```



Sprites (8/26)

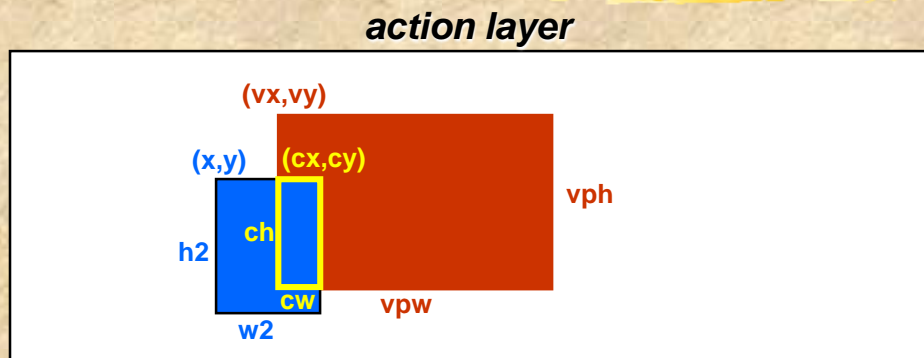
- Για να γίνεται γρήγορα η εκτύπωση των sprites θα πρέπει τα sprites να είναι **sorted ανά z-ordering** (π.χ. αυξανόμενου z-ordering) και να εκτυπώνονται με την αντίστοιχη σειρά
- **η ταξινόμηση γίνεται κατά τη δημιουργία των sprites** και συνεπάγεται ότι **διατηρείται ένα priority list** από sprites
- είναι **σπάνιο να χρειάζεται αλλαγή του z-ordering** ενός sprite κατά την εκτέλεση, αλλά και αν πρέπει να υποστηριχθεί δεν είναι πολύπλοκη λειτουργία η μετακίνηση στη λίστα



Θα το “ακούσετε” και σαν z-sorting, z-buffering ή και depth sorting, ορολογία που είναι «δανεισμένη» από 3D GFX.

Sprites (9/26)

■ Moving, z-ordering and display (4/5)

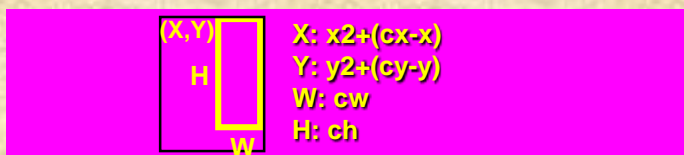


X: $dx+(cx-vx)$
Y: $dy+(cy-vy)$

view window



current frame



Θέση του ορατού τμήματος στο film bitmap

•Το $\langle cx, cy, cw, ch \rangle$ είναι σε pixel coordinates όλου του action layer (όπως και το $\langle vx, vy, vw, vh \rangle$).

•Είναι το *intersection rectangle* μεταξύ του (sprite) frame box και του (terrain) view window.

•Το $\langle dx, dy, dw, dh \rangle$ είναι σε screen buffer coordinates.

•Τα frames είναι σε film bitmap coordinates.



Sprites (10/26)

```
template <class T> bool clip_rect(  
    T x, T y, T w, T h,  
    T wx, T wy, T ww, T wh,  
    T* cx, T* cy, T* cw, T* ch  
) {  
    *cw = T(std::min(wx + ww, x + w)) - (*cx = T(std::max(wx, x)));  
    *ch = T(std::min(wy + wh, y + h)) - (*cy = T(std::max(wy, y)));  
    return *cw > 0 && *ch > 0;  
}  
  
bool clip_rect (const Rect& r, const Rect& area, Rect* result) {  
    return clip_rect(  
        r.x,  
        r.y,  
        r.w,  
        r.h,  
  
        area.x,  
        area.y,  
        area.w,  
        area.h,  
  
        &result->x,  
        &result->y,  
        &result->w,  
        &result->h  
    );  
}
```




Sprites (11/26)

```
// generic clipper assuming any terrain-based view
// and any bitmap-based display area

class Clipper {
public:
    using View = std::function<const Rect&(void)>;

private:
    View    view;

public:
    Clipper& SetView (const View & f)
        { view = f; return *this; }

    bool    Clip (
        const Rect&    r,
        const Rect&    dpyArea,
        Point*         dpyPos,
        Rect*           clippedBox
        ) const;

    Clipper (void) = default;
    Clipper (const Clipper&) = default;
};
```



Sprites (12/26)

```
bool Clipper::Clip (const Rect& r, const Rect& dpyArea, Point* dpyPos, Rect* clippedBox) const {  
  
    Rect visibleArea;  
    if (!clip_rect(r, view(), &visibleArea))  
        { clippedBox->w = clippedBox->h = 0; return false; }  
    else {  
        // clippedBox is in 'r' coordinates, sub-rectangle of the input rectangle  
        clippedBox->x = r.x - visibleArea.x;  
        clippedBox->y = r.y - visibleArea.y;  
  
        clippedBox->w = visibleArea.w;  
        clippedBox->h = visibleArea.h;  
  
        dpyPos->x = dpyArea.x + (visibleArea.x - view().x);  
        dpyPos->y = dpyArea.y + (visibleArea.y - view().y);  
  
        return true;  
    }  
}
```

$X: dx+(cx-vx)$
 $Y: dy+(cy-vy)$



Sprites (13/26)

```
void Sprite::Display (Bitmap dest, const Rect& dpyArea, const Clipper& clipper) const {  
  
    Rect    clippedBox;  
    Point   dpyPos;  
  
    if (clipper.Clip(GetBox(), dpyArea, &dpyPos, &clippedBox)) {  
  
        Rect clippedFrame {  
            frameBox.x + clippedBox.x,  
            frameBox.y + clippedBox.y,  
            clippedBox.w,  
            clippedBox.h  
        };  
  
        MaskedBlit(  
            currFilm->GetBitmap(),  
            clippedFrame,  
            dest,  
            dpyPos  
        );  
  
    }  
}
```

A display method that clips correctly within terrain bounds without making Sprite class dependent on the terrain implementation classes



Sprites (14/26)

The Clipper class and the Mover functor encapsulate the terrain-specific clipping and motion filtering

```
const Clipper MakeTileLayerClipper (TileLayer* layer) {  
    return Clipper().SetView(  
        [layer](void)  
        { return layer->GetViewWindow(); }  
    );  
}  
  
const Sprite::Mover MakeSpriteGridLayerMover (GridLayer* gridLayer, Sprite* sprite) {  
    return [gridLayer, sprite](const Rect& r, int* dx, int* dy) {  
        // the r is actually always the sprite->GetBox():  
        assert(r == sprite->GetBox());  
        gridLayer->FilterGridMotion(r, dx, dy);  
        if (*dx || *dy)  
            sprite->SetHasDirectMotion(true).Move(*dx, *dy).SetHasDirectMotion(false);  
    };  
};
```


Sprites (15/26)

Σε τέτοιες περιπτώσεις χρησιμοποιούμε **uniform** frame boxes για films και rendering, με ground baseline, και εάν θέλουμε ακρίβεια σε collision έχουμε συνοδευτικά **minimal BBs** / frame

■ Frame change (1/3)

- Ήδη έχουμε παρουσιάσει την **SetFrame** για το βασικό Sprite class. Τι παραπάνω χρειάζεται?
- Η αλλαγή ενός frame μπορεί να συνεπάγεται αλλαγή μεγέθους ενός sprite.
- Αλλά η σχεδίαση του animation μπορεί να είναι τέτοια που απαιτεί κατάλληλη ευθυγράμμιση των διαφόρων frames
 - ◆ με τρόπο διαφορετικό από το να θεωρούμε το πάνω αριστερό σημείο του sprite πάντα σταθερό κατά την εναλλαγή των frames





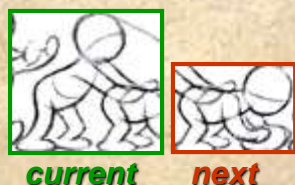
Sprites (16/26)

■ *Frame change (2/3)*

- Πρέπει να μπορούμε να ορίζουμε κατά την αλλαγή των frames το σταθερό οριζόντιο και κάθετο άξονα σε περίπτωση που το νέο frame έχει διαφορετικές διαστάσεις από το προηγούμενο
 - ◆ αυτό ονομάζεται **frame-change alignment policy - FCAP**
- Αυτό σημαίνει ότι το sprite για να αντεπεξεχθεί στο εκάστοτε FCAP ενδέχεται να χρειαστεί να αλλάξει το origin του frame.
- Έχουμε τις εξής περιπτώσεις για το FCAP
 - ◆ Οριζόντια σταθερή πλευρά (Left, Right, Center)
 - ◆ Κάθετη σταθερή πλευρά (Top, Bottom, Center)


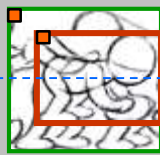
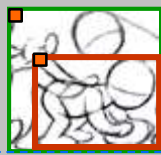
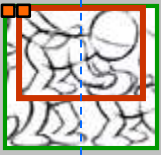
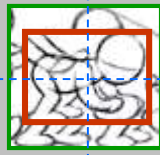

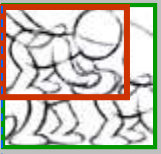
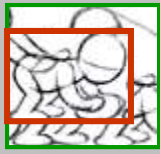

Sprites (1726)

■ Frame change (3/3)



• Το είδος του frame change alignment μπορεί να ποικίλει ανά animation, αλλά «για το καλό σας» φροντίστε να μην ποικίλει και από frame σε frame στο ίδιο animation (αυτό σημαίνει να κατευθύνετε τους γραφίστες να μην κάνουν κάτι τέτοιο).

• Θα χρειαστεί να ορίσετε τους δυο enum τύπους, να εισάγετε δύο επιπλέον attributes στο sprite class (hFrameAlign και vFrameAlign) και να τροποποιήσετε την SetCurrFrame να κάνει κατάλληλα offset το (x,y) του sprite.

		Vertical alignment		
		Top	Center	Bottom
Horizontal alignment	Right			
	Center			
	Left			

→ hang-on-rope characters

→ flying characters

→ walking characters



Sprites (18/26)

■ Gravity (1/5)

- Αρκετοί χαρακτήρες του παιχνιδιού ενδέχεται να μην «γνωρίζουν» να πετούν
- αλλά και αυτοί που το «γνωρίζουν», εάν δεν κάνουν την ενέργεια που απαιτείται για να πετάξουν ενώ βρίσκονται στον αέρα, σίγουρα πέφτουν,
 - ◆ εκτός και εάν είναι χαρακτήρες που σχεδιάστηκαν να μην υφίστανται το νόμο της βαρύτητας.
- Ό τρόπος που προγραμματίζουμε την αντίδραση ενός sprite instance στην περίπτωση που δεν «πατάει» σε solid terrain, δηλ.
 - ◆ σε θέση «κάτω» από την οποία βρίσκεται solid tile
 - ◆ ή έστω στα solid pixels ενός slope tile
- είναι συνήθως η ενεργοποίηση ενός falling animation (...αρκετές φορές συνοδευόμενο και από το χαρακτηριστικό συνεχές ουρλιαχτό...).



Sprites (19/26)

■ Gravity (2/5)

- Χρειάζονται οι εξής επεκτάσεις:
 1. Εισαγωγή ενός boolean flag για το εάν ο χαρακτήρας είναι «εθισμένος» στη βαρύτητα, **bool gravityAddicted**, το οποίο μπορεί να ρυθμίζεται at run-time
 - π.χ. τη στιγμή ενός *jump / fly animation* μπορεί να απενεργοποιήσουμε τη βαρύτητα για ένα χαρακτήρα
 2. Εισαγωγή ενός boolean flag για το εάν ο χαρακτήρας αυτή τη στιγμή είναι σε falling state, **bool isFalling**,
 - το οποίο είναι **true** εάν **gravityAddicted == true** και ο χαρακτήρας δεν πατάει πάνω σε *solid terrain*
 3. Εισαγωγή μίας callback (Notifier pattern) για την περίπτωση ακριβώς που ο χαρακτήρας μεταβαίνει από falling σε non-falling state και αντίστροφα.
 4. Επέκταση της Move ώστε να εφαρμόζει έλεγχο «βαρύτητας» για τον χαρακτήρα μετά την κίνηση του



Sprites (20/26)

A generic *Sprite* and *GridLayer* agnostic gravity handler

■ Gravity (3/5)

```
class GravityHandler {
public:
    using OnSolidGroundPred      = std::function<bool(const Rect&)>;
    using OnStartFalling         = std::function<void(void)>;
    using OnStopFalling          = std::function<void(void)>;
protected:
    bool          gravityAddicted = false;
    bool          isFalling = false;
    OnSolidGroundPred onSolidGround;
    OnStartFalling   onStartFalling;
    OnStopFalling    onStopFalling;
public:
    template <typename T> void SetOnStartFalling (const OnSolidGroundPred & f)
        { onStartFalling = f; }
    template <typename T> void SetOnStopFalling (const T& f)
        { onStopFalling = f; }
    template <typename T> void SetOnSolidGround (const T& f)
        { onSolidGround = f; }
    void Reset (void) { isFalling = false; }
    void Check (const Rect& r);
};
```



Sprites (21/26)

■ Gravity (4/5)

Gravity check triggering the required callbacks which are responsible to handle the transition

```
void GravityHandler::Check (const Rect& r) {  
    if (gravityAddicted) {  
        if (onSolidGround(r)) {  
            if (isFalling) {  
                isFalling = false;  
                onStopFalling();  
            }  
        }  
        else  
            if (!isFalling) {  
                isFalling = true;  
                onStartFalling();  
            }  
    }  
}
```



Sprites (22/26)

■ Gravity (5/5)

Extending the Sprite class for gravity and linking grid layers with sprite gravity outside the Sprite class, while enabling direct motion (unchecked) in case some animation require free motion without any tests

```
class Sprite {
    bool
    GravityHandler
    GravityHandler&
    Sprite&
    bool
    Sprite&

    directMotion = false;
    gravity;
    GetGravityHandler (void)
        { return gravity; }
    SetHasDirectMotion (bool v) { directMotion = true; return *this; }
    GetHasDirectMotion (void) const { return directMotion; }
    Move (int dx, int dy) {
        if (directMotion) // apply unconditionally offsets!
            x += dx, y += dy;
        else {
            quantizer.Move(GetBox(), &dx, &dy);
            gravity.Check(GetBox());
        }
        return *this;
    }
};

void PrepareSpriteGravityHandler (GridLayer* gridLayer, Sprite* sprite) {
    sprite->GetGravityHandler().SetOnSolidGround(
        [gridLayer](const Rect& r)
            { return gridLayer->IsOnSolidGround(r); }
    );
}

sprite->SetHasDirectMotion(true).Move(dx, dy).SetHasDirectMotion(false); ←Instant move
```




Sprites (23/26)

■ Collision engagement (1/4)

- Είχαμε δει ότι ακόμη και με optimizations με τη χρήση compressed bit masks για τα bitmaps, το collision detection είναι αρκετά χρονοβόρο.
- Εάν έχουμε σε μία σκηνή N χαρακτήρες, όλοι οι δυνατοί συνδυασμοί για collision detection θα απαιτούσαν $N \times (N-1)/2$ ελέγχους μεταξύ των sprites
 - ◆ π.χ. για 50 sprites θα ήθελα 1225 ελέγχους ανά game loop, κάτι που προφανώς θα έχει σοβαρές αρνητικές επιπτώσεις στο game frame rate
- Όμως δεν είναι όλοι αυτοί οι έλεγχοι αναγκαίοι στην πλοκή ενός παιχνιδιού, π.χ.
 - ◆ διακοσμητικοί χαρακτήρες μπορεί ποτέ να μην ελέγχονται,
 - ◆ πολλές εκτοξευμένες σφαίρες δεν χρειάζεται να ελεγχθούν ως προς την πιθανή μεταξύ τους σύγκρουση,
 - ◆ συγκεκριμένα αντικείμενα μπορεί να ελέγχονται μόνο για σύγκρουση με συγκεκριμένες κατηγορίες χαρακτήρων



Sprites (24/26)

■ Collision engagement (2/4)

Έστω μία σκηνή του παιχνιδιού τη χρονική στιγμή t στην οποία έχω τα παρακάτω

- 20 διακοσμητικούς χαρακτήρες
- 15 εκτοξευμένα αντικείμενα από τον παίκτη με στόχο τρεις (3) «κακούς» χαρακτήρες
- 30 εκτοξευμένα αντικείμενα από τους τρεις (3) «κακούς» προς τον παίκτη
- 1 παίκτη

Αρκεί να ελέγξω για collision μεταξύ κάποιων από τα:

- 15 αντικείμενα $\leftarrow \rightarrow$ 3 κακοί, δηλ. 45 ελέγχους
- 30 αντικείμενα $\leftarrow \rightarrow$ παίκτης, δηλ. 30 ελέγχους

Συνολικά χρειάζομαι 75 ελέγχους εάν γνωρίζω τη λογική του παιχνιδιού, αλλιώς θα χρειάζομαι τον απαγορευτικό αριθμό των 2145 ελέγχων!

- Για το σκοπό αυτό ορίζουμε μία ειδική singleton κλάση η δίνει τη δυνατότητα να ορίζονται τα ζευγάρια από sprite για τα οποία πρέπει να γίνει έλεγχος και η οποία αναλαμβάνει να καλεί σε κάθε game loop το collision detection μόνο γι' αυτά.
 - Το ζεύγος (A,B) θεωρείται ότι είναι το ίδιο με το (B,A) και δεν επιτρέπουμε registration και των δύο
 - Αυτά ονομάζονται **collision pairs**



Sprites (25/26)

■ Collision engagement (3/4)

```
class CollisionChecker final {
public:
    using Action = std::function<void(Sprite* s1, Sprite* s2)>;
    static CollisionChecker singleton;

protected:
    using Entry = std::tuple<Sprite*, Sprite*, Action>;
    std::list<Entry> entries;
    auto Find (Sprite* s1, Sprite* s2) -> std::list<Entry>::iterator;

public:
    void Register (Sprite* s1, Sprite* s2, const Action& f)
        { assert(!In(s1,s2)); entries.push_back(std::make_tuple(s1, s2, f)); }
    void Cancel (Sprite* s1, Sprite* s2);
    void Check (void) const;

    static auto GetSingleton (void) -> CollisionChecker&
        { return singleton; }
    static auto GetSingletonConst (void) -> const CollisionChecker&
        { return singleton; }
};
```




Sprites (26/26)

■ Collision engagement (4/4)

```
auto CollisionChecker::Find (Sprite* s1, Sprite* s2) -> std::list<Entry>::iterator {
    return std::find_if(
        entries.begin(),
        entries.end(),
        [s1, s2](const Entry& e) {
            return std::get<0>(e) == s1 && std::get<1>(e) == s2 ||
                std::get<0>(e) == s2 && std::get<1>(e) == s1;
        }
    );
}

void CollisionChecker::Cancel (Sprite* s1, Sprite* s2) {
    entries.erase(Find(s1,s2));
}

void CollisionChecker::Check (void) const {
    for (auto& e: entries)
        s1 → if → std::get<0>(e) → CollisionCheck(std::get<1>(e))
        action → std::get<2>(e) ( std::get<0>(e), std::get<1>(e) );
}
```



Sprites extra (1/3)

- Ο ρόλος των fields *typeId* και *stateId* είναι πολύ σημαντικός
- Χρησιμοποιούνται για να χαρακτηρίσουν γενικά τον τύπο ενός sprite καθώς και την κατάσταση στην οποία βρίσκεται
 - το typeId είναι immutable ενώ το stateId αλλάζει
- είναι σημαντικά για τη λογική αντίδρασης σε περίπτωση collision ή σε μερικές περιπτώσεις και για animation
- **μπορείτε να έχετε hash table με όλα τα sprites ενός είδους σε έναν πίνακα**



Sprites extra (2/3)

```
class SpriteManager final {
public:
    using SpriteList    = std::list<Sprite*>;
    using TypeLists     = std::map<std::string, SpriteList>;
private:
    SpriteList          dpyList;
    TypeLists           types;
    static SpriteManager singleton;
public:
    void                Add (Sprite* s); // insert by ascending zorder
    void                Remove (Sprite* s);
    auto                GetDisplayList (void) -> const SpriteList&
                        { return dpyList; }
    auto                GetTypeList (const std::string& typeId) -> const SpriteList&
                        { return types[typeId]; }
    static auto         GetSingleton (void) -> SpriteManager&
                        { return singleton; }
    static auto         GetSingletonConst (void) -> const SpriteManager&
                        { return singleton; }
};
```

Call these on sprite
ctor/ dtor (assume
type is set initially
as a constructor
argument)



Sprites extra (3/3)

```
#define  ALIEN_TYPE  "alien"
#define  BULLET_TYPE  "bullet"

extern void KillAlien (Sprite* alien);

void CreateBullet (const Point& pos, AnimationFilm* film) {

    Sprite* bullet = new Sprite(pos.x, pos.y, film, BULLET_TYPE);

    // TODO: prepare here all the visual staff

    auto& aliens = SpriteManager::GetSingleton().GetTypeList(ALIEN_TYPE);

    for (auto* alien : aliens)
        CollisionChecker::GetSingleton().Register(
            bullet,
            alien,
            [](Sprite* alien, Sprite* bullet)
                { KillAlien(alien); }
        );
};
```

Can get all sprites of a given type and apply collision registration as required. This avoids separate bookkeeping