Dr. Mileff Péter

GRAFIKA PROGRAMOZÁSA

GYAKORLATI 2D GRAFIKA I. RÉSZ

Miskolci Egyetem
Általános Informatikai Tanszék

A MEGJELENÍTÉS ALAPJAI…

(Egyszerű megjelenítés - szoftveres)

Általános áttekintés

- A 2D megjelenítés két dimenziós képi elemekkel (textúrákkal), objektumokkal dolgozik.
- **A 2D grafikus motor feladata a képszintézisben:**
  - sorra vegye a memóriában tárolt objektumokat
  - elvégezze a megfelelő 2D transzformációkat
    - Eltolás, elforgatás, nyújtás, stb
  - kirajzolja a látható részeket a képernyőre
- A végleges kép az objektumok kombinációjaként jön létre
  - Bizonyos objektumok takarhatják egymást
  - Bizonyos objektumok részei átlátszók lehetnek

Általános áttekintés

- **A textúrák/képek fizikai elhelyezkedése:**
  - **Szoftveres render:** minden esetben a központi memóriában
  - **GPU renderelés:** vagy a központi, vagy a videó memóriában
    - Implementáció függő
      - Korai programok inkább a központi memória: **glBegin/glEnd**
      - Modern szoftverek inkább a videó memória: **VBO**
- A tárolás formája: tömb
- A tömbök színinformációkat tartalmaznak az objektumokról,
  - méretük a textúra minőségétől függ.

Általános áttekintés

- **A mai grafikai szoftverek jellemzői:**
  - főként 32 bites színmélységű képekkel dolgoznak
    - 4 byte – RGBA, ARGB
  - 1 textúra felbontása akár 1024x768 pixel is lehet.
    - Fizikailag a kártya többet is tud, de nem jellemző a sok nagyméretű textúra
- **Színcsatornák alapján kép csoport:**
  - alfa csatornával rendelkező és – RGB (24 bit)
  - nem rendelkező képi elemek – RGBA (32 bit)
- **A megkülönböztetés fontos:**
  - a megjelenítési eljárás, a gyorsítási lehetőségek a két csoportnál eltérnek.

Miért egyszerű a megjelenítés?

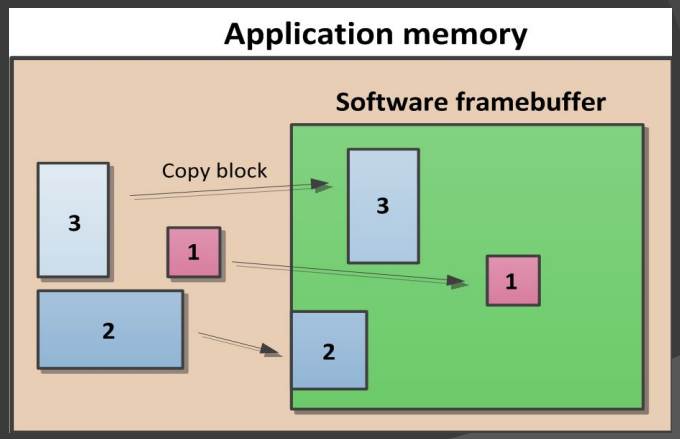
- Az egyszerű raszterizáció a szoftveres megjelenítésre épül:
  - csak az objektum képét alkotó alacsony szintű byte-okkal, pixelekkel foglalkozunk
  - A képek tárolása a központi memóriában történik
    - szokványos két dimenziós tömbben
  - **A megjelenítés:**
    - a tömböket kirajzoljuk egy szoftveres megvalósítású framebufferbe
    - majd átadjuk a videokártyának megjelenítésre
  - **Miért egyszerű?**
    - **lényegében egy memóriamásolási művelet:** az objektum tömbjét átmásoljuk a framebufferbe
    - Nincs dimenzió leképzés, vetítés vagy egyéb transzformáció

RENDERELÉS ALFA CSATORNA NÉLKÜL…

Renderelés alfa csatorna nélkül

- **Csak RGB színkomponens! Miért jó?**
- **Bármely két objektum egymásra rajzolható:**
  - Nincs szükség az egymás alatt lévő objektumok színének összemosására
  - Egyszerűbb és gyorsabb kirajzolási algoritmus:
    - Nem kell (félig) átlátszó pixelekkel foglalkozni
    - Az adatok mozgathatók egy blokként a videó bufferbe
    - Kevesebb adat mozgatás: 24 bit
- **Gyors raszterizáció célja:**
  - meg kell próbálni minden műveletet a lehető legnagyobb blokkra kiterjedő módon elvégezni,
  - ezzel elkerülhető a felesleges adatmozgatás és számítás

Renderelés alfa csatorna nélkül



Textúrák másolása blokként a framebufferbe

Renderelés alfa csatorna nélkül

- **A kirajzolás ebben az esetben azt jelenti:**
  - a központi memória blokkjait a framebuffer meghatározott területére mozgatjuk
    - a memóriamásolás (pl.) műveletével
    - Pl. C++ - *memcpy(), Java - System.arraycopy*
    - Nagyságrendi sebességnövekedés érhető el, mert az adatok egyben mozognak.
- **A módszer így nem teljes értékű:**
  - A blokkorientált adatmozgatás nem kezeli a kilógó objektumokat!
- Megoldás: szegmentálni kell a másolandó blokkot a képernyőn látható tartalom függvényében,
  - ha valamilyen irányban kilógna az objektum

Renderelés alfa csatorna nélkül

- A szegmentálás függ a framebuffer tömb tárolási formájától:
  - Soronként, vagy oszloponként
    - A gyakorlatban soronkénti a domináns.
- Ez soronkénti szegmentációt jelent.
- A renderelés során textúrát soronként másoljuk a célterületre.
- Így lehetővé válik a képernyőről kilógó részek levágása.
- Bár ez további számításokat igényel, a nagyságrendi teljesítmény növekedés így is megmarad
  - Az alfás rendereléshez képest

Blit Texture példa

```
void BlitTexture(CVector position) {

if (position.x + mTexture->width < 0 || position.x >
    gGraphics->GetFrameBuffer()->GetWidth())
  {
    return;
  }

  if (position.y + mTexture->height >
   gGraphics->GetFrameBuffer()->GetHeight() || position.y < 0)
  {
    return;
  }
```

Blit Texture példa

```
CFramebuffer *fbuffer = gGraphics->GetFrameBuffer();
for(unsigned int j=0; j < m_pTexture->height; j++) {
    fbuffer->BlitArray(mTexture->texels+(j*mTexture->width*4),
    mTexture->width*4,position.x,position.y+j);
}
}
```

BlitArray függvény:

```
void BlitArray(uint32_t* image_block, int length, int x, int y)
{
   int offset = y * m_Width + x;
   if (offset < screen_buffer_size)
      memcpy(m_pFrameBuffer+offset,image_block,length);
}
```


Renderelés alfa csatorna nélkül

- A módszert kevésbé szokás alkalmazni ma már
  - mert az alfás képek / effektek dominálnak
- **Főként a régi játékokban dominált a módszer**
  - gyenge CPU, nincs GPU
  - Tipikus példa erre az oldalra scrollozós játékok (Prince of Persia, Giana Sisters, Gods, stb)
  - próbálták a grafikai elemeket úgy rajzolni, hogy minimális legyen az átlátszó pixelek száma

RENDERELÉS ALFA CSATORNÁVAL…

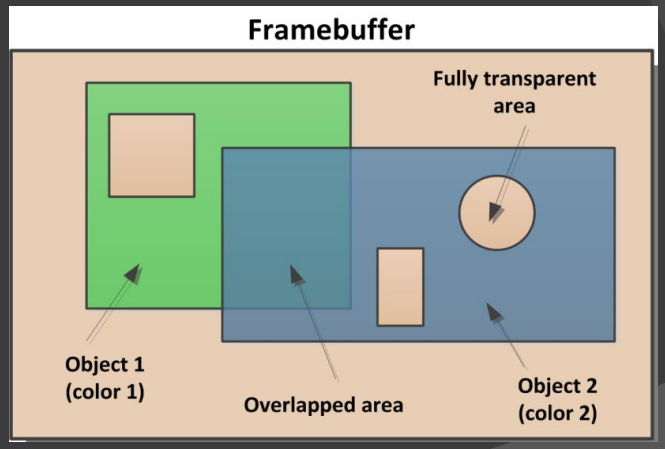
Renderelés alfa csatornával

- **A raszterizáció igazi kihívása.**
  - **RGBA színkomponens – 32 byte**
- **Oka:**
  - az objektumok rendelkeznek átlátszó részekkel
  - átfedhetik egymást
- **Megoldás:**
  - nincs más lehetőség, a textúrákat tartalmazó tömböket pixelenként kell végigiterálni
  - Meg kell vizsgálni, hogy szükséges-e pixel összemosás valamelyik réteggel.
- A grafikus motor tehát az objektumokat reprezentáló képi elemeken pixelenként halad végig és készíti el a képet.

Renderelés alfa csatornával

- **A renderelés hátránya:**
  - általában sok elemet kell kirajzolni a képernyőre,
    - ezek szintén sok pontból állhatnak – jobb minőség végett.
  - A pixelenkénti rajzolás több ezer függvényhívással és redundáns számítással jár.
    - Minden pixel esetén külön ki kell olvasni a memóriából a képpont színét,
    - majd a környezeti adatok függvényében pedig meghatározni a képernyőn való pozícióját,
    - és elvégezni a szín framebufferbe való írását
    - pl. pFrameBuffer[y * screenWidth + x] = color
  - **Összességében:** nagy adathalmaz pixelenként való iterálása - **lassú**

Renderelés alfa csatornával



- **Átlátszó és átfedő objektumok**

Renderelés alfa csatornával

- **Szakirodalom elnevezése:**
  - **„blittelés” – blitting**
- Minden mai ablakkezelő rendelkezik ilyen jellegű függvénnyel:
  - Pl. Microsoft – BitBlt, Linux Xlib – XcopyArea.

Példa

```
void RenderAlphaImageSoftware(Vector2 position) {
  // X irányú láthatóság vizsgálata
  if (position.x + texture.width < 0 || position.x > screen.width)  {
      return;
  }

  // Y irányú láthatóság vizsgálata
  if (position.y + texture.height > screen.height || position.y < 0) {
      return;
  }

 for i = 0 to texture.width
   for j = 0 to texture.height
      pixelIndex = j * texture.width + i;
            pixel = texture.pixels[pixelIndex];
```

Példa

```
// Nem rajzolunk, ha teljesen átlátszó
if ( pixels.alfa == 0 )
   continue;

// Rajzolunk, ha teljesen nem átlátszó
if ( pixels.alfa == 1 ){
      framebuffer.SetPixel(position.x + i, position.y + j, pixel);
      continue;
         }

      // Lekérdezük a framebufferben lévő színt
frameBufferPixel = framebuffer.GetPixel(position.x + i, position.y + j);

      // Az új szín a kettő kombinációja lesz
   framebuffer.SetPixel(position.x + i, position.y + j, frameBufferPixel  + pixel);
}
```


Mi a probléma az egyszerű megjelenítéssel?

- **Az objektumok foroghatnak és méretük is változhat!**
- Ezek újabb problémákat vetnek fel:
  - A tömböket valós időben transzformálni kell
    - Gondoljunk a forgatásra!
    - megnő a memória műveleteket száma
    - csökken a teljesítmény
- **Probléma áthidalása a korai játékokban:**
  - a forgatási és egyéb fázisokat külön-külön megrajzolták.
  - Példa:
    - amikor egy űrhajó elfordult, akkor a forgatási szögnek megfelelő képet rajzolták ki a tényleges elfogatás helyett

POLIGON ALAPÚ MEGJELENÍTÉS…

Poligon alapú raszterizáció

- A GPU-k által leginkább alkalmazott megoldás
- **Lényege röviden:**
  - Minden objektum modellje háromszögekből épül fel,
    - akár három, akár kétdimenziós modellről van szó.
  - Két dimenziós esetben: a textúra két darab háromszögre kerül ráfeszítésre.
- **Renderelés:**
  - a GPU vagy CPU sorra veszi a memóriából a háromszögeket és raszterizálja azokat valamilyen algoritmussal.
  - Leggyakrabban az úgynevezett scanline algoritmust használják a kép előállítására,
    - lineáris interpolációt felhasználva képpontonként

Poligon alapú raszterizáció

- A mai GPU hardver tipikusan ennek a folyamatnak a támogatására fejlődött ki.
- A textúra és a háromszögek a GPU memóriában tárolódnak
- Nincs szükség adatmozgatása a központi memória és a GPU memóriája között.
- Jelenleg az OpenGL és a DirectX implementációkkal készült szoftverek mind ezt a megoldást alkalmazzák.

OBJEKTUMOK MOZGATÁSA…

2D Objektumok mozgatása

- „Mozgó textúrák” jobb elnevezése: **Objektum**
  - Több, mint pusztán egy textúra
  - Egyéb tulajdonságai vannak:
    - Pl. látható-e vagy sem, mozgatható, forgási iránya, stb
  - A játékiparban előnyösen használják az elnevezést
    - Vagy valamilyen rokon értelmű megfelelőjét
- **Egy objektum mozgatása:**
  - az alakzat (jelen esetben egy kép) változtatja a pozícióját.
  - Valamilyen esemény hatására. Pl. egérmozgás
  - A pozíció változásnak irányvektora és sebessége van, amelyek meghatározzák a mozgás jellegét.

2D Objektumok mozgatása

- **Matematikailag megfogalmazva:**

  objektum új pozíció(x,y) = aktuális pozíció (x,y) + sebesség(v)*irányvektor(x,y)

- **Folyamatos mozgás:**
  - Minden frame-ben minden objektumra elvégezzük a fenti műveletet,
  - így a mozgás folyamatos lesz.
  - Ha az irányvektor nullvektor, akkor az objektum megáll.

2D Objektumok mozgatása

Program fő ciklus (main loop):

```
While(!exit){
 HandleEvents();
 MoveObjects()
 DrawObjects();
}
```

A MoveObjects() függvény (lényege):

```
for (int i=0;i<numOfObjects;i++){
   CVector2 oldpos = obj[i].pos;
   // A direction egy CVector2 osztály típus
   obj[i].pos = oldpos + speed*direction;
}
```


2D Objektumok mozgatása

- **Előnye:**
  - Egyszerű megoldás
- **Hátrányok:**
  - A bemutatott megoldás nem hatékony.
  - **Probléma:** akkor jelentkezik, amikor nagyon különböző sebességű számítógépekkel dolgozunk.
  - Lassú gép esetén a mozgás sebessége lassú lesz,
  - Gyors számítógép esetén pedig túl gyors.
  - Korábbi játékok esetén tipikusan megfigyelhető jelenség volt.
    - Főleg a DOS korszak

ELTELT IDŐ ALAPÚ MOZGÁS...

(TIME BASED MOVEMENT)

Eltelt idő alapú mozgás

- Klasszikus megoldás módosított változata(i)
- Biztosítja az objektumok azonos mozgatási sebességét
  - különböző sebességű gépeken is
- **Az algoritmus háttere:**
  - Minden grafikai motor belül rendelkezik egy fő ciklussal
    - main loop, game loop
  - Gyors számítógépen ez a ciklus gyorsabban, a lassú gépen pedig lassabban hajtódik végre.
  - **Cél:** mérjük meg két fő ciklus között eltelt időt
    - kapunk egy olyan tényezőt, amely felhasználható a gépek közötti sebesség egységesítésére.
  - nagyobb felbontású (legalább milliszekundum) órával

Eltelt idő alapú mozgás (Példa)

```
while( game_is_running ) {
    prev_frame_tick = curr_frame_tick;
    curr_frame_tick = GetTickCount();
    elapsed_time = curr_frame_tick – prev_frame_tick;
    update( elapsed_time);
    render();
}
```

GetTickCount() függvény:

a rendszer elindítása óta eltelt időt adja vissza milliszekundumban

Eltelt idő jellemzői

- Lekérdezése minden esetben operációs rendszer függő.
- Ideális esetben egy 0 és 1 közé eső dupla pontosságú lebegőpontos szám.
  - Pl.: 0.003568
- Ha értéke nulla, akkor a timer felbontása nem elég kicsi.
  - Két frame között nem tudja mérni az időt
- **Nulla érték nem használható!**
  - **Oka:** hogy a tényező szorzóként fog szerepelni a mozgásoknál

```
obj[i].pos = oldpos + elapsed_time*(speed*direction);
```

Eltelt idő jellemzői

- A szorzó tényező a pozíció additív tagjára van hatással.
- **Gyors gépen ez az idő kicsi:**
  - így az additív tag így kisebb lesz.
  - folyamatosabb mozgást fogunk tapasztalni.
- **Lassabb gépeken ez az érték nagyobb:**
  - a mozgás kevésbé folyamatos
    - emberi szemmel talán nem is észrevehető
  - de a megtett út azonos lesz a gyors számítógépen futtatott verzióval!

Eltelt idő kiegészítés 1

- **1. Eltelt idő értékének maximalizálása**
  - **Probléma:** a gép „beakad” ha az operációs rendszerben bizonyos háttérfolyamatok több erőforrást használnak fel
    - az eltelt idő megnő, amely egy nagyobb ugrást eredményez a mozgatott objektumnál.
  - **Tipikus példa a debuggolás:** a szoftvert megállítjuk debuggolás céljából,
    - újraindítva az eltelt idő értéke nagyon nagyot ugrik ha nem maximáljuk
  - **Cél:** érték maximalizálása. pl. 1.0 értékre

Eltelt idő kiegészítés 2

- **2. Eltelt idő értékének „simítása”**
  - **Probléma:** az eltelt idő értéke két grafikailag azonos terheltségű ciklus között megugorhat.
  - A szoftverben ez legtöbbször nem okoz gondot,
  - Célszerű azonban ezt kompenzálni!
  - Pl. átlagot számolni a korábbi és az új ciklus eltelt idejére:

```
elapsed_time += curr_frame_tick – prev_frame_tick;
elapsed_time *= 0.5;
```

- Bár a kiegészítések hatékonyak, de nem tökéletesek.
- Bizonyos esetekben célszerűnek tartják az FPS minimum illetve maximális számát is rögzíteni.

OBJEKTUMOK ANIMÁCIÓJA…

Objektumok animációja

- Az animáció fontos szerepet tölt be a számítógépes grafikában.
- Ettől válik igazán „élővé” a szoftver
  - Pl. menü, ablak vagy egy ugráló figura animációja
- **A klasszikus animáció:** egy textúrahalmaz megadott szekvenciában történő váltogatása bizonyos sebességgel.
- **Textúrahalmaz:** a textúrák egy tömbje, amely tartalmazza az animáció egyes fázisait.
- A gyakorlatban a textúrákból álló objektumot **Sprite**-nak is nevezik.
- Minél több a fázis, annál folytonosabb lesz a megjelenítéskor az objektum animációja.

Példa megvalósítás

```
class CSprite {
        string mName                        // Sprite name
    vector<CSpriteFrame*> mFrames;    // Frames vector
    unsigned int mNumFrames;              // Number of frames
    unsigned int mActualFrame;            // Actual frame
    CVector2 mPosition;           // position of the sprite
    CVector2 mScale;              // Sprite scale value
    unsigned int mLastUpdate;  // The last update time
    unsigned int mFps;            // The number of frames per second
        float mZRotation;             // Z axis rotation value
public:
...
};
```

Példa megvalósítás

- **CSprite osztály:** kompakt egység, amely egy animációs szekvenciát tárol
- **Elemei:**
  - **A sprite neve:** fontos, mert névvel hivatkozni sokkal egyszerűbb
    - Pl. kérem az “ugrás” animációt
  - **CSpriteFrame osztály:** egy frame-et tárol
    - Ezekből képzett vektor reprezentálja az animációt
  - Pozíció, méret, elforgatás
  - Fázisok száma, aktuális fázis azonosítója
  - Animáció sebessége

Példa megvalósítás

```
class CSpriteFrame {
   CTexture2D* mFrame;                                  /// Frame texture
   CString mName;                                       /// Name of the frame
    vector<CBoundingBox2D*> mBBoxOriginal;              /// Original Bounding boxes
    vector<CBoundingBox2D*> mBBoxTransformed; /// Transformed Bounding boxes

public:

   /// Default Constructor
   CSpriteFrame();


   /// Copy operator overloading function
   CSpriteFrame& operator= (const CSpriteFrame& _spriteFrame);
…
};
```

Példa megvalósítás

- **CSpriteFrame osztály:**
  - Tényleges frame képének tárolása: **CTexture2D**
  - Frame neve: néha szükség lehet rá
    - névvel hivatkozás egyszerűbb!
  - Befoglaló doboz: ütközés vizsgálathoz
    - eredeti: fontos a megtartása a számítások gyorsításához
    - transzformált: az objektum elforgatott, stb befoglaló doboza

Példa megvalósítás

```
class CTexture2D {
    CVector2 mPosition;
    CVector2 mRotation;
    CVector2 mScale;
    bool bVisible;
    CVAOobject* mTextureVAO;    // Storage data in VAO
    sColor mColor;              // Color information
    string mFilename;           // Holds the filename of the texture
    string mName;               // Name
    float mWidth;               //  Stores the width of the texture
    float mHeight;              // Stores the height of the texture
    unsigned int mTextureID;    // Holds the texture ID
    int mID;                    // Global ID of the texture
…
};
```

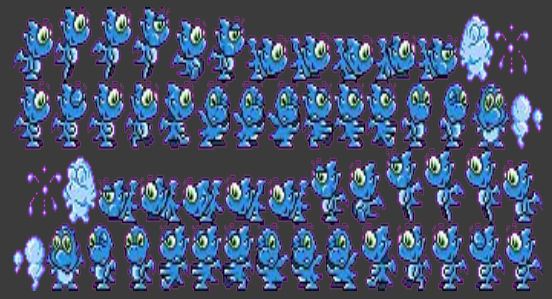
Példa megvalósítás

- **CTexture2D osztály:**
  - Pozíció, elforgatás, méret
  - Fájlnév
  - Textúra neve
  - Szín információ
  - Azonosítók:
    - OpenGL azonosító: egyedi textúra ID az OpenGL-től
    - Globális ID az engine-ben
  - Vertex és textúra koordináták tárolása - VAO-ban

Animációk tárolása a fájlrendszerben

- A fájlrendszerben az animáció képei többféle módon tárolhatók
  - **Spritesheet** megoldás
    - **A leginkább elterjedtebb megoldás:**
      - egy nagyobb képben tároljuk az egyes frame-eket egymás mellett.
  - Külön kép minden frame-nek
    - talán könnyebb feldolgozás

Példa animáció tárolásra



Spritesheet

Aladdin játék (SNES)

Klasszikus spritesheet

- **Az animációkat egymás mellé helyezve tárolják**
- Korai spritesheet megoldásban:
  - A készítők kiválasztanak valamilyen egységes háttér színt
    - így tudják mit nem kell megjeleníteni - **colorkey**
- Ma már alfa csatornás képekben tárolják
- **A fázisok mérete változhat:**
  - a betöltés során a betöltőnek képesnek kell lennie valamilyen logika alapján részekre vágni,
    - majd ezeket a részeket külön textúrába szervezni
    - ehhez azonban tudunk kell a fázisok méreteit
      - Ha a fázisok külön fájlban lennének, akkor ez nem gond
  - Szükség van tehát valamilyen kiegészítő leíró fájlra!

Sprite leíró fájl

- **Mit ad meg egy leíró fájl?**
  - megadja, hogy az egyes grafikai elemek mely pixel pozíciókon foglal helyet
  - a befoglaló dobozok pontos mérete
    - egy fázis akár több box-al is rendelkezhet
  - Esetleg a fázis neve
    - néha meg kell különböztetni a speciális frame-eket
  - lehetőség van bármi egyéb fontosnak vélt adat tárolására
- **Milyen formátum?**
  - Célszerű valamilyen olyan ismert tárolási formát választani mint a például a JSON, vagy az XML

Egy komoly játék tehát igényel valamilyen leírót!

Sprite leíró fájl - példa

```
<animation name="My sprite anim">
    <frames numoffframes="2">
       <frame name="Robot Anim 1" file="robot1.tga">
              <aabb minx="0" miny="0" maxx="64" maxy="64" />
            </frame>
            <frame name="Robot Anim 2" file="robot2.tga">
              <aabb minx="0" miny="0" maxx="70" maxy="60" />
          </frame>
    </frames>
</animation>
```


Sprite alapú animáció

- Az ilyen jellegű kétdimenziós rajzokat összefoglaló néven „**Pixel Art**”-nak nevezik
- **Oka:** nagyrészt kézzel készülnek pixelről pixelre rajzolva
  - ez nehéz, időigényes folyamat
- Ma a mobil eszközökre készített játékok nagy része ebbe a kategóriába tartozik

Sprite alapú animáció

- A technikával nagyon komoly minőségű, úgynevezett **cinematic** típusú játékok is megvalósíthatók
- **Jellemzője:**
  - a nagyon folytonos, gördülékeny animáció
  - sok, akár több száz fázisból elkészítve
  - Az animációk felvehetők digitalizálva is:
    - Neve: “Rotoscoping”

Híres játékok:

Prince of Persia (1989), Flashback (1992), Aladdin (1993), Lion King (1994), Heart of Darknes (1998)

Heart of Darkness (1998)

Jellemzője: profi munka

- rengeteg cinematikus elem
- sok fázis, gördülékeny animáció

ANIMÁCIÓK KIRAJZOLÁSA…

Animációk kirajzolása

- **Az animáció megvalósítása:** nem más, mint a különböző framek egymás utáni kirajzolása.
- Figyelembe kell venni az animáció sebességét.
- Nem rajzolhatjuk ki minden frame-ben a következő fázist, mert akkor az animáció nagyon gyors lesz.
- **Szükség van tehát:**
  - az animáció sebességének megadására
  - annak figyelembe vételére a kirajzolási folyamatban
- Ennek támogatására ismét az időt nyújt segítséget!

Animációk kirajzolása

```
/** Update frames */
void CSprite::Update(){
    uint32_t ticks = GetOSTicks();
    // Dontes az animacio valtasarol
    if ( 1000.0f/m_iFps < (ticks - m_iLastUpdate) ){
        m_iLastUpdate = ticks;
        if (++m_iActualFrame > m_iNumFrames){
            m_iActualFrame = 1;
        }
    }
}
```


Animációk kirajzolása

- **Példa magyarázata:**
- *m_iFps* az a sebesség, amely elvárunk az animált objektumtól.
- A megoldás láthatóan egyszerű:
  - az *1000.0f/m_iFps* értéke megadja, hogy 1 másodpercben hányszor kell végrehajtani az animáció váltást.
  - Amikor az eltelt idő meghaladja ezt az értéket, válthatunk a következő fázisra.

GAMEOBJECT OSZTÁLY…

GameObject osztály

- Önmagában a **Sprite** osztály még nem elég!
- **Felhasználhatjuk:**
  - Pl. GUI elemek (pl. Animált gombok, stb) létrehozására, vagy tényleges játékra szánt objektumok alapjául.
- **Nem teljes:**
  - Egy két dimenziós játékban egy játék objektumnak nem csak 1 animációs fázisa van,
  - Külön-külön lehet minden állapotához.
  - Nem tartalmaz egyéb, fontos tulajdonságokat!
- **Célszerűen:** a Sprite-ok tömbjét kell alkalmazni
  - Ahol az objektum állapotától függően (séta, guggolás, stb) ezek válthatók.
- Nevezhetjük **GameObject2D**-nek.

GameObject osztály

- Fontos az objektumok kirajzolásának sorrendje!
  - Bizonyos helyzetekben az objektumok átfedhetik egymást
  - Pl. vannak olyan objektumok (pl. Felhő), amely rárajzolódik például a hely objektumra
- Ez valamilyen programozói logika által meghatározott sorrendet jelentenek.
- **Megvalósítás:** megkövetel egy numerikus érték bevezetését
  - a sorrendiséget reprezentálni kell.
  - pl. *z* érték

GameObject osztály

Egy megvalósítási logika lehet a következő:

- Minél kisebb z értékkel rendelkezik az objektum, annál közelebb van a nézőhöz,
  - azaz annál később kerül kirajzolásra.
- A megvalósítás megköveteli az objektumok *z* értéke alapú rendezését
  - a kirajzolás megfelelő sorrendje így biztosítható.

Példa megvalósítás

```
class CGameObject2D {
    CVector2              m_vPosition;              // Position of the object
    CVector2              m_vNewPosition;           // Position of the object
    vector<CSprite*>      m_Animations;             // Animation
    CVector2              m_vDirection;             // Direction of the movement
    float                 m_fSpeed;                 // Speed of the object
    bool                  m_bVisible;               // Visible or not
    bool                  m_bCollidable;            // Collidable or not
   unsigned int           m_uiCurrentAnim;          // Current Animation Frame
    unsigned int          m_uiNumberOfFrames; // Number of Animations
    unsigned int          ID;                       // ID of the Object
    int                   m_iZindex;                // z index of the object
public:
...
};
```