Dr. Mileff Péter

# GRAFIKA PROGRAMOZÁSA

## GYAKORLATI 2D GRAFIKA II. RÉSZ

Miskolci Egyetem
Általános Informatikai Tanszék

## OPTIMALIZÁLT MEGJELENÍTÉS ALAPJAI…



## Általános áttekintés

- A vizualizáció alapvető problémája a **sebesség**!
  - Lassú programokat senki sem szereti.
- **Két összetevője:**
  - **1. Megfelelő végrehajtó hardver:** minden szoftvernek megvan a minimum és a javasolt igénye
  - **2. Megfelelő implementáció:**
    - Gyakori probléma, hogy a hardverileg elégséges gépen is lassú a vizualizáció
    - **Oka:** a nem hatékony megvalósítás. Lassú algoritmusok.
    - A CPU és GPU megfelelő kihasználása
    - Algoritmusok optimalizálása
- Emellett vannak kiegészítő technikák
  - a mai négyteljesítményű CPU és GPU mellett is alkalmazni kell



## Általános áttekintés

- Számtalan optimalizációs lehetőség van
  - Sokszor ezek egyediek a szoftverre nézve
- **Sebesség alapú optimalizálás fő szabálya:**
  - csak azt kell kirajzolni, ami valóban is látszik
  - minimalizálni kell a felesleges felülrajzolásokat.
- A két dimenziós megjelenítés előnye:
  - sokkal egyszerűbb megoldások, mint a háromdimenziósban
  - az alkalmazott technikák matematikailag könnyebben érthetőbbek.

# BEFOGLALÓ OBJEKTUM ALAPÚ MEGJELENÍTÉS…

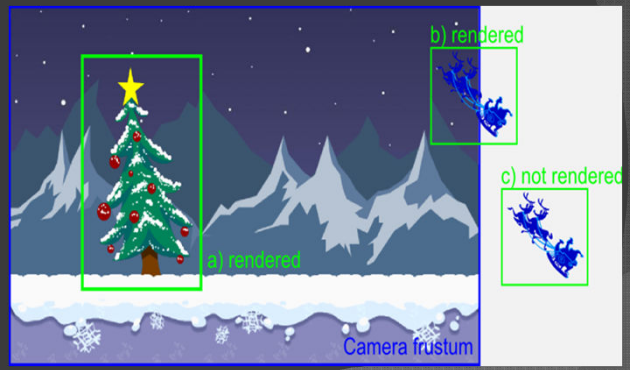
## Befoglaló objektum alapú megjelenítés

- A két dimenziós (és a 3D is) játékok többségében alkalmazott megoldás
  - befoglaló objektum segítségével redukálják a kirajzolási adatokat
  - Gyorsítanak algoritmusokat.
- Mi az általános logika mögötte?
  - Bármely vizualizációs feladatot célszerű nagyobb egységekben kezelni
    - Pl. Rajzolás, ütközés vizsgálat, stb
  - A nagyobb egység jelentős gyorsulást eredményez

## Befoglaló objektum alapú megjelenítés

- **Befoglaló objektumok technikája:**
  - Minden objektumnak meg kell határozni (vagy megadni) az őt minimálisan befoglaló entitást
    - legegyszerűbb esetben téglalap, vagy kör,
    - bonyolultabb esetben pedig poligonnal szokás megadni
  - A gyakorlatban leginkább a befoglaló téglalapot, azaz más néven a **befoglaló dobozt** szokták alkalmazni.
    - Bounding Box
- **Az algoritmus logikája:**
  - Az objektumok megjelenítése során:
    - minden kirajzolási fázis előtt megvizsgáljuk, hogy az adott objektum befoglaló doboza benne van-e a képernyő tartományában
    - {0-szélesség, 0-magasság} – 2D Kamera frustum

## Befoglaló objektum alapú megjelenítés

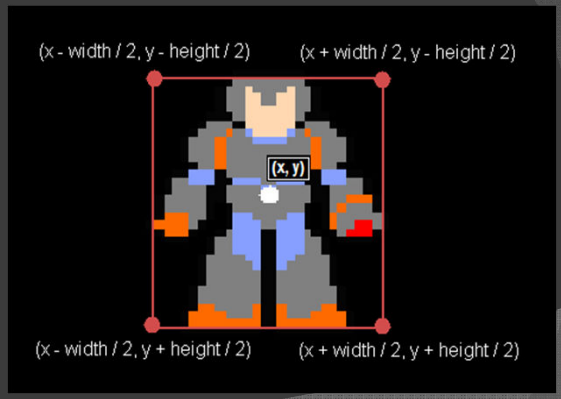


# Befoglaló objektum alapú megjelenítés

- **Befoglaló doboz jellemzői:**
  - törekednek arra, hogy a legjobban illeszkedőt állapítsák, vagy adják meg.
    - Általában a Sprite betöltésekor határozzák meg
  - **Oka:** elkerüljék az olyan hibás számításokat, mint például a következőt:
  - **Példa:**
    - a BB tartalmaz jobb oldalt néhány átlátszó pixelt.
    - Az objektum úgy helyezkedik, hogy csak ezek a pixelek lógnak be a képernyőbe.
    - Ekkor az objektum megjelenítésre fog kerülni annak ellenére, hogy nem látszik belőle semmi.
    - Átmegy a grafikus API csővezetékén, transzformálódik, azaz erőforrást használ



# Minimális befoglaló doboz




A doboz általában megegyezik a kép szélességével és magasságával



# Befoglaló objektum alapú megjelenítés

**Miért pont doboz vagy kör?**

- Mert nagyon egyszerű elemek
- A velük való későbbi számolások:
  - egyszerűek, kis számításigényűek
    - Pl. ütközésvizsgálat, forgatás, eltolás, stb
- Bár nem közelítik jól az objektumot, mégis hatékonyak és jól alkalmazhatók a gyakorlatban.



# Példa BB osztály

```
/// 2D Axis Aligned Bounding Box
class CBoundingBox2D{
    CVector2 minpoint;          // Box minpoint
    CVector2 maxpoint;          // Box maxpoint
    CVector2 bbPoints[4];       // bounding box points
    float boxHalfWidth;         // box half width
    float boxHalfHeight;        // box half height
    matrix4x4f tMatrix;         // Transformation matrix
 public:
...
};
```

## Példa BB osztály jellemzői

- Két dimenzióban egy befoglaló dobozt 4 ponttal lehet megadni,
  - A doboz 4 sarka
- A későbbi számítások gyorsítása érdekében:
  - célszerű tárolni a képernyő koordináta rendszeréhez viszonyított minimum, illetve maximum pontját.
    - ez a bal felső és jobb alsó pontot jelenti általában
  - Célszerű tárolni a fél szélesség, fél magasság értékeket is

13

## Befoglaló doboz jellemzői

- Az objektum mozgatása (eltolás, forgatás, nyújtás) során a doboz koordinátáját szintén transzformálni kell.
  - Ehhez nyújt segítséget az osztályban a mátrix adattag.
- **A számítás ideje:**
  - **1. Az objektum mozgásakor** kiszámítjuk annak új pozícióját.
    - Minden frame-ben megtörténik, minden objektumra
  - **2. Igény esetén:** a BB pontjait akkor számítjuk ki, amikor a program használni fogja azt
    - pl. ütközésvizsgálat, képernyő vágás, stb,
    - Probléma: ha több helyen kell használni, a többszöri kiszámításhoz több erőforrásra van szükség.

14

## BB alapú megjelenítés

- Annak eldöntése, hogy az objektum megjeleníthető-e vagy sem:

```
if (bb->maxpoint.x < 0 || bb->minpoint.x > screen_width ||
    bb->minpoint.y > screen_height || bb->maxpoint.y < 0)
{
      return false;
}
```

15

## BEFOGLALÓ DOBOZOK FORGATÁSA…

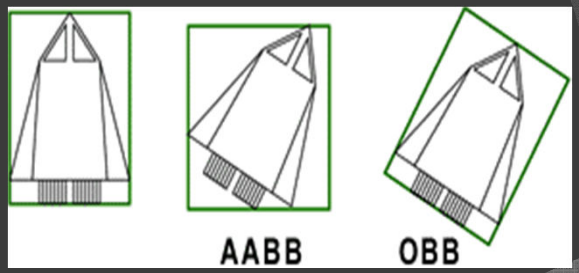
16

## Befoglaló doboz jellemzői

- Az objektumok forgatása során a dobozt is forgatni kell
  - A sarkokat manuálisan transzformálni kell
- **Ez alapján két csoport:**
- **Axis-Aligned Bounding Boxes (AABB):**
  - olyan téglatest (2d-ben téglalap), amelynek minden éle egy koordinátatengellyel párhuzamos.
- **Oriented Bounding Box (OBB):**
  - olyan téglatest, amely az objektum forgatásával együtt fordul.

17

## AABB és OBB




18

## Befoglaló doboz a gyakorlatban

- A gyakorlatban az AABB megvalósítása sokkal egyszerűbb, mint az OBB esetében.
- **Lényegesen könnyebb AABB esetében:**
  - az ütközésvizsgálat,
  - a dobozok átfedésének kiszámítása,
  - a képernyővel való vágás kiszámítása,
  - egyéb számítások
- **Negatívum:**
  - minden forgatáskor újra kell számolni a doboz pontjait
  - Így elforgatáskor a pontossága romlik

19

## Befoglaló doboz a gyakorlatban

- Az AABB pontjainak újraszámolása három lépésen:
  - forgatás esetén transzformáljuk a doboz pontjait,
  - megkeressük a minimális és a maximális pontokat,
  - majd a pontok alapján létrehozzuk az új dobozt
- Az OBB esetében:
  - a fenti pontok közül elegendő csak az első lépést végrehajtani.
  - A nehézség abban rejlik, amikor meg kell állapítani, hogy két doboz átfedi-e egymást.
    - Matematikailag komplexebb
- Az alkalmazott megoldás mindig a készítendő szoftver igényeitől függ.
  - Legtöbb esetben az AABB bőven elegendő.

20

# 2D ÜTKÖZÉSVIZSGÁLAT…



## Ütközésvizsgálat

- A játékprogramok elengedhetetlen eleme az objektumok egymással való interakciója
  - annak a vizsgálata, hogy két objektum mikor ütközik egymásnak, mikor érintkeznek.
- Nem csak a játékok világára jellemző:
  - ugyanezen elveket alkalmazzuk akkor is, amikor például az egeret egy menüelem felé helyezzük.
- Természetesen a számítógépes játékokban ennek domináns szerepe van
  - a játékélmény ezen interakciók hatására alakul ki
    - Pl. egy akciójátékban a lövedék eltalálja az ellenséget



## 2D ütközésvizsgálat a gyakorlatban

**Az ütközésvizsgálat lényege:**

- valahogyan algoritmikusan érzékelni kell, hogy két vagy több objektum két dimenziós képe átfedi egymást.
- **A valós ütközésvizsgálat:** azt jelenti, hogy egy objektum egy pixele átfedi egy másik objektum pixelét.
- Ennek érzékelése számításigényes!
- A játékfejlesztők így hamisítanak:
  - valamilyen objektumba próbálják meg befoglalni a mozgatott elemeket
    - Befoglaló doboz, kör, poligon, stb
  - és erre elvégezni az ütközések vizsgálatát így redukálva a számításigényt.



## 2D ütközésdetektálás a grafikus motorban

- **Ütközések érzékelésének ideje:**
  - objektumok és a befoglaló dobozának új pozícióba való mozgatás **előtt**
    - Különben az objektum belelóg a „falba”, összeragadhatnak
  - Minden objektumot minden objektummal meg kell vizsgálni.
- **Algoritmus:**
  - Mozgatás során ki kell számolni az objektum és az ő befoglaló dobozának új pozícióját.
  - Az ütközésvizsgálatot erre az új értékekre kell végrehajtani.
  - Amennyiben nem ütközik úgy felveheti az új pozíciót,
  - különben pedig el kell dönteni mi legyen az objektummal
    - pl. megáll, felrobban, stb

## 2D ütközésdetektálás a grafikus motorban

- **Célszerű két vektort is alkalmazni**
  - 1. vektor az új pozíciónak
  - 2. vektor a réginek.
- Ugyanis ha megáll az objektum, úgy régi értéket kell meghagyni

25

## 2D ütközésdetektálás általánosan

```
void checkCollisions() {
   // check other sprite's collisions
   spriteManager.resetCollisionsToCheck();
   // check each sprite against other sprite objects.
   for (Sprite spriteA : spriteManager.getCollisionsToCheck()) {
      for (Sprite spriteB : spriteManager.getAllSprites()) {
         if (handleCollision(spriteA, spriteB)) {
            // The break helps optimize the collisions
            //  The break statement means one object only hits another
            break;
         }
      }
   }
```

26

## BEFOGLALÓ DOBOZ ALAPÚ ÜTKÖZÉSVIZSGÁLAT…

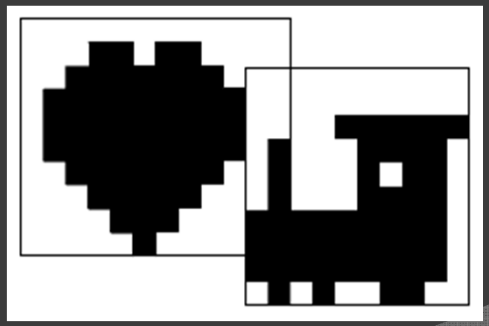
27

## Befoglaló doboz alapú ütközésvizsgálat

- Az évek során több különféle ütközésvizsgálati technika alakult ki:
  - pl. Separate Axis Theorem
- Legnépszerűbb és egyszerűbb a befoglaló doboz alapú technika
  - Általánosan: rectangular collision detection
- **Lényege:**
  - Az elv azonos az objektumok képernyőn való megjelenítésének vizsgálatával.
  - Amikor két objektum befoglaló doboza (vagy esetleg köre) átfedi egymást, az objektumok ütköznek.

28

## Befoglaló doboz alapú ütközésvizsgálat



- A befoglaló dobozok átfedéséből egyértelműen meghatározható az ütközés ténye.

29

## Egy lehetséges algoritmus

- A gyors ellenőrzés miatt célszerű azt érzékelni mikor nincs ütközés
  - így felesleges számításoktól kíméljük meg a CPU-t

```
bool checkCollision(CBoundingBox2D* boxObj1, CBoundingBox2D* boxObj2){

if (boxObj1->GetMaxPoint()->x < boxObj2->GetMinPoint()->x ||
boxObj1->GetMinPoint()->x > boxObj2->GetMaxPoint()->x){
        // Nincs ütközés
        return false;
}
 if (boxObj1->GetMaxPoint()->y < boxObj2->GetMinPoint()->y ||
boxObj1->GetMinPoint()->y > boxObj2->GetMaxPoint()->y){
        //nincs utkozes
        return false;
}
   return true;
}
```

30

## Befoglaló doboz alapú megoldás hibái

- **„Lyukas” objektumok ütközése:**
  - objektumok átlátszó résszel
  - Ha valójában a lyukas részek fedik csak át egymást, úgy nem történik tényleges ütközés!
- A hiba ellenére a játékfejlesztés területén ez a megoldás terjedt el leginkább.
- **Oka:**
  - az egyszerűsége és a redukált számításigény
  - a legtöbb játék esetében gyors mozgás közben nem vesszük észre, hogy „nem is az objektum pixelével ütköztünk”.

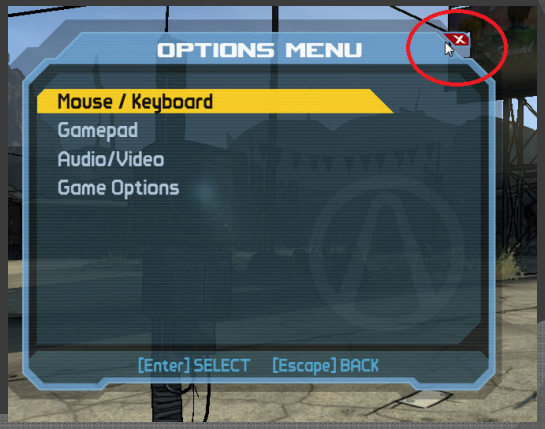
31

## Példa ismert „hibákra”

- A mai modern játékok menürendszere
  - pl. BorderLands - Unreal Engine, Crysis sorozat, stb
  - Szinte minden játék esetében
- **Érzékelés:**
  - Egy nem szabályos, például rombusz alakú gomb kiválasztása
  - Az alsó, nem valós területére mozdítva az egeret a felhasználói interakció megtörténik (a gomb kivilágít).

32

## Borderlands - Menü




## A hiba orvosolása

- Kialakult egy nagyon egyszerű, de könnyen megvalósítható megoldás a gyakorlatban.
- A befoglaló doboz méretét nem pixelre pontosan az objektum képére számolják ki
- Redukálják annak méretét valamilyen értékkel.

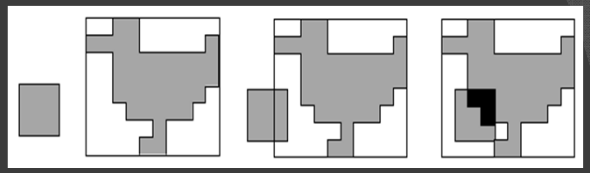
```
object->width = <ACTUAL BITMAP WIDTH>;
object->height = <ACTUAL BITMAP HEIGHT>;
object->col_width = object->width * 0.80;
object->col_height = object->height * 0.80;
object->col_x_offset = (object->width - object->col_width) / 2;
object->col_y_offset = (object->height - object->col_height) / 2;
```

## PIXEL SZINTŰ ÜTKÖZÉSVIZSGÁLAT…

## Pixel szintű ütközésvizsgálat

- **Elnevezés:** per-pixel collision detection
- Valódi, pontos ütközésvizsgálat
- Minden szoftver esetében a pixel alapú megoldás lenne az ideális
- Számításigénye nagy!
- Emiatt azonban csak ott alkalmazzák, ahol erre kimondottan igény van.

## Pixel szintű ütközésvizsgálat



- **Bal oldal:** nincs ütközés
- **Jobb oldal:** valós ütközés
  - Mivel a pixelek takarják egymást, a befoglaló dobozok is
- **Középső:**
  - a „golyó" még nem hatolt be a kacsa „testébe", viszont a mintáját tartalmazó téglalap alakú tartományba már igen
  - A befoglaló doboz vizsgálat már ütközést jelez!

37

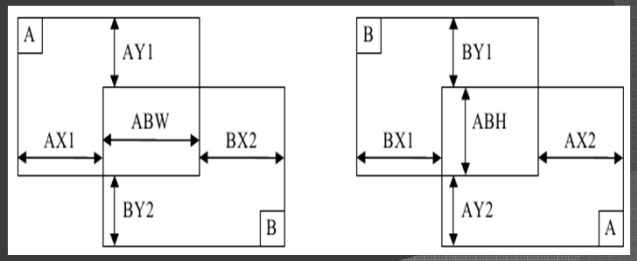
## Pixel szintű ütközésvizsgálat

- **A helyes megoldás algoritmusa:**
  - meg kell vizsgálni, hogy a két objektum területének vannak-e egymást fedő pontjai.
  - **Triviális megoldás:** a két objektum minden pixelét megvizsgáljuk
    - Számításigényes, sok ciklus
  - **Optimális megoldás:** a befoglaló doboz alapján.
    - Ha a két terület nem érintkezik, a két objektumnak nem lehet egymást fedő átlátszatlan pontja,
    - Ha egymásba lóg az objektumok doboza, meg kell vizsgálni a közös részt
    - Ehhez végig kell pásztázni annak pontjait
    - Ha találunk legalább egy olyan helyet, ahol mindkét objektum átlátszatlan, akkor ütköztek.

38

## Pixel szintű ütközésvizsgálat

- Felhasználjuk a befoglaló dobozok által átfedett területet
- Csak ezen a területen belüli pixeleket kell átvizsgálni




39

## Pixel szintű ütközésvizsgálat

- **Az algoritmus:**
  - Az algoritmusnak az ABW és ABH területet kell pixelenként átvizsgálni
  - Addig míg nem talál mindkét objektum képi leképzésénél legalább egy darab nem átlátszó pixelt.
- Mi okozza nagy számítási igényt?
  - A dupla ciklus, ami végighalad a sprite-ok képpontjain.
  - Minden pont értékét a központi memóriából le kell kérni, majd összehasonlítani egymással.
  - Kis méretű sprite-ok esetén ez nem okoz nagy gondot,
  - Nagyobb méret esetén jelenős erőforrást igényel
    - dupla ciklusba ágyazott feltételes utasítás végrehajtása minden megjelenítési frame-ben.

40

## Pixel szintű ütközésvizsgálat (példa)

```
for (i=0; i < over_height; i++) {
        for (j=0; j < over_width; j++) {
                if (pixel1 > 0) && (pixel2 > 0) return true;
                pixel1++;
                pixel2++;
            }
            pixel1 += (object1->width - over_width);
            pixel2 += (object2->width - over_width);
    }
```

41

## EGYÉB KIEGÉSZÍTŐ MEGOLDÁSOK…

42

## Kiegészítő megoldások

- Az irodalomban számos egyéb megoldás is kialakult
  - Általában szoftverre specifikus eljárások
- **Példa:** a **bitmaszk alapú pixeles ütközésvizsgálat**
- **Lényege:**
  - a megoldásnál egy fekete fehér képét készítik el az objektumnak
    - pl. pálya ahol mehet a motor
  - Ütközésvizsgálat esetén ezt a 0 és 1 értéket tartalmazó bitképet vizsgálják.
- **Előnye:**
  - bitek jelzik az ütközési területet,
  - így kevesebb helyet foglalnak el a memóriában, mintha RGBA kép lenne
    - RGBA esetén egy integer-ként tárolva egy pixelt tudunk feldolgozni, a bitmaszk alapú megvalósításnál 4 darabot

43

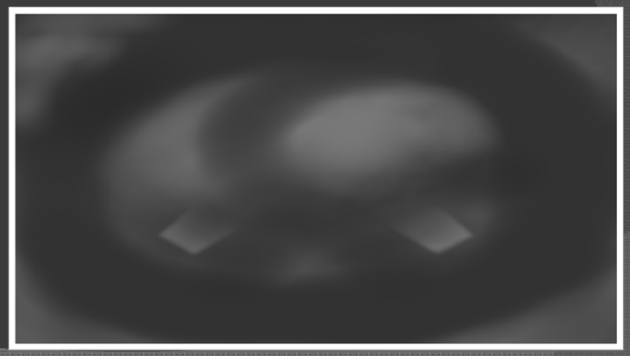
## Bitmaszk alapú ütközésvizsgálat (példa)

- Normál játéktér

44

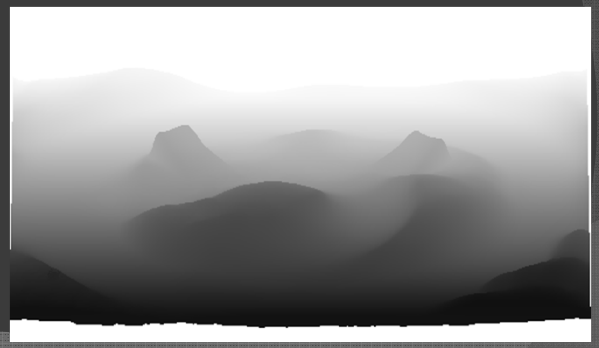
## Bitmaszk alapú ütközésvizsgálat (példa)

- Objektumok mozgástere fekete fehér textúraként



## Bitmaszk alapú ütközésvizsgálat (példa)

- Mélység térkép a játéktérhez



## Bitmaszk alapú ütközésvizsgálat (példa)

- Valós játékmenet


GAME OVER