Production Systems and Information Engineering
Volume 12 (4), pp. 22-39
doi: https://doi.org/10.32968/psaie.2024.4.2

ÜTKÖZÉS DETEKTÁLÁS 2D JÁTÉKOKBAN

MILEFF PÉTER
Miskolci Egyetem, Informatikai Intézet,
Általános Informatikai Tanszék
peter.mileff@uni-miskolc.hu

BEDNARIK LÁSZLÓ
Miskolci Egyetem, Informatikai Intézet,
Általános Informatikai Tanszék
laszlo.bednarik@uni-miskolc.hu

Abstract. Az ütközés érzékelés a játékfejlesztés egyik alapvető aspektusa, mivel lehetővé teszi, hogy a játékobjektumok reális és intuitív módon "kommunikáljanak" egymással. A 2D-s játékokban az ütközés érzékelés általában a játékobjektumok befoglaló dobozai vagy alakzatai közötti átfedések ellenőrzésével történik. Azonban a játékok bonyolultabbá válásával kifinomultabb algoritmusok és technikák fejlődtek ki. Jelen cikk áttekintést nyújt a játékokban a 2D ütközések érzékelésére használt különféle algoritmusokról és technikákról. Bemutatásra kerülnek a különböző megközelítések előnyei és hátrányai, mint például a koordináta tengelyekkel párhuzamos befoglaló doboz (AABB), az orientált befoglaló doboz, az elválasztó tengelyek elmélete (SAT) és más hasznos technikák. Ezenkívül megvitatjuk e megközelítések kihívásait és korlátait, és útmutatást adunk a megfelelő megközelítés kiválasztásához a konkrét igények és korlátok alapján.

Keywords: Ütközés érzékelés, befoglaló doboz, SAT

1. Bevezetés

A modern játékfejlesztés egy dinamikus és gyorsan fejlődő terület, amely magában foglalja a legmodernebb interaktív élmények létrehozását a játékosok számára a platformok és eszközök széles skáláján. A hardver- és szoftvertechnológia fejlődésével a játékfejlesztők most olyan hatékony eszközökhöz és motorokhoz férhetnek hozzá, amelyek lehetővé teszik számukra, hogy rendkívül valósághű grafikát, magával ragadó hangzásvilágot és kifinomult játékmechanikát hozzanak létre. A modern játékfejlesztés egyik fő trendje a többplatformos fejlesztés térnyerése, amely lehetővé teszi a játékok több eszközön és platformon történő lejátszását, beleértve a konzolokat, PC-ket és mobileszközöket. Ez új lehetőségeket nyitott meg a játékfejlesztők előtt, hogy szélesebb közönséget érjenek el, és olyan innovatív játékélményeket teremtsenek, amelyeket a játékosok számos eszközön élvezhetnek [2].

A játékfejlesztés egyik fontos aspektusa az ütközésérzékelés, amely az a folyamat, amely érzékeli, ha egy játékvilágban két vagy több objektum érintkezik vagy metszi egymást [1]. Az ütközésérzékelés elengedhetetlen a valósághű és magával ragadó játékélmény megteremtéséhez. Legyen szó első személyű lövöldözős játékról, versenyjátékról, platformerről vagy kirakós játékról, az ütközésérzékelést használják annak biztosítására, hogy a játékvilág az elvárásoknak megfelelően viselkedjen, és a játékosok értelmes módon kommunikálhassanak tárgyakkal és karakterekkel.

Az ütközésérzékelés a modern játékfejlesztésben összetett folyamat, amely megköveteli mind a játékvilágban lévő tárgyak fizikai tulajdonságainak, mind a viselkedésük szimulálására használt számítási technikák mély megértését [3]. Az ütközésérzékelésnek többféle megközelítése létezik, mindegyiknek megvannak a maga erősségei és gyengeségei.

A cikk központi témája és célja, hogy áttekintést adjon a legfontosabb technikákról, amelyek elengedhetetlenek egy kétdimenziós játék létrehozásához. A gyakorlati szempontból bemutatott algoritmusok segíthetik az ütközésérzékelés hatékony megvalósítását.

2. Ütközés detektálás

A játékprogramokban alapvető fontosságú az objektumok kölcsönhatásának kezelése, vagyis annak meghatározása, hogy két objektum mikor ütközik vagy érintkezik egymással. Ez az elv nem csupán a játékokra jellemző, hanem például a menüelemek egérrel való kiválasztásakor is alkalmazzuk. A számítógépes játékok esetében az interakciók pontos érzékelése különösen fontos, mivel a játékélmény nagymértékben ezen interakciók alapján alakul ki. Például egy akciójátékban a lövedék eltalálhatja az ellenfelet, vagy megakadályozhatja, hogy a főhős áthaladjon a labirintus falán.

Az ütközésvizsgálat lényege, leegyszerűsítve, az, hogy algoritmusok segítségével meg kell határozni, hogy két vagy több objektum kétdimenziós képe átfedi-e egymást. Pontosabban, azt kell ellenőrizni, hogy van-e olyan pixel az egyik objektumban, amely átfedi a másik objektum pixelét.

A játékfejlesztés során fontos döntést kell hozni az ütközésdetektáló rendszer kiválasztásáról. Ez nem mindig egyszerű, mivel a játékok interakciói gyakran komplexek, és sokszor előre nem látható problémákat rejthetnek. Az alkalmazott modell azonban jelentős hatással van a fejlesztési időre és a játékélményre. Alapvetően az alábbi két csoportba sorolhatjuk az ütközésdetektáló rendszereket [4][5][6]:

- **Pixel alapú ütközésvizsgálat:** az ütköző objektumokhoz tartozó képek pixeleinek átfedését vizsgálja. Precíz, valós ütközést képes érzékelni

- **Befoglaló objektum alapú ütközésdetektálás:** Az objektumok átfedését nem pixel szinten, hanem valamilyen befoglaló objektum(ok) (doboz, kör, poligon, stb) szintjén határozzuk meg. Általában nem precíz ütközést tesz lehetővé.

A pixel alapú ütközésdetektálás számításigényes és bonyolult lehet, különösen akkor, ha az objektum textúrája összetett. Emiatt a játékfejlesztők gyakran próbálnak egyszerűsített alakzatokat, például köröket vagy négyzeteket alkalmazni az ütközések vizsgálatára. Ezek az egyszerű formák könnyen kezelhetők és a velük végzett műveletek – mint az ütközésvizsgálat, forgatás, és eltolás – sokkal kevésbé számításigényesek [7], mint egy bonyolultabb poligon vagy pixel szintű vizsgálat. Bár ezek az egyszerű alakzatok nem mindig követik pontosan az objektumok formáját, hatékonyak és jól alkalmazhatók a gyakorlatban.

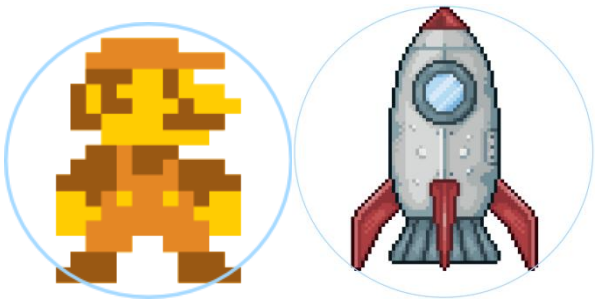
Jelen cikk a továbbiakban a befoglaló objektum alapú ütközési megoldásokra koncentrál. Bemutatja az ennek megvalósításhoz szükséges legfontosabb

eljárásokat.

3. Befoglaló kör alapú ütközés

3.1. Befoglaló kör alapú ütközés

A befoglaló kör (*Bounding Circle*) (esetleg ellipszis) alapú ütközésvizsgálat a lehető legegyszerűbb ismert megoldás annak eldöntésére, hogy két objektum fedésben van-e [8]. Ilyenkor minden objektumot egy (valamilyen szinten illeszkedő) körbe foglalunk be. A kör középpontját és sugarát általában annak megfelelően határozzuk meg, hogy az ütközésvizsgálat a lehető leghatékonyabb legyen. Ha túl nagy a kör vagy nem jól illeszkedik az objektumra, akkor az objektum szélein nem valós ütközések lesznek. Ha pedig túl kicsi, akkor az objektum akár részegesen belelóghat a másik objektumba, például a falba. Természetesen a kört lehetséges akár automatikusan is számolni. Ilyenkor az objektum mögötti grafikai elem (kép) betöltése után, annak szélessége és magassága adhat támpontot. Az automatizmus azonban sok esetben nem megfelelő, mert a befoglaló kör meghatározásánál nem mindig az a cél, hogy az objektum minden pixele bele essen. Az alábbi ábra ez mutatja be:

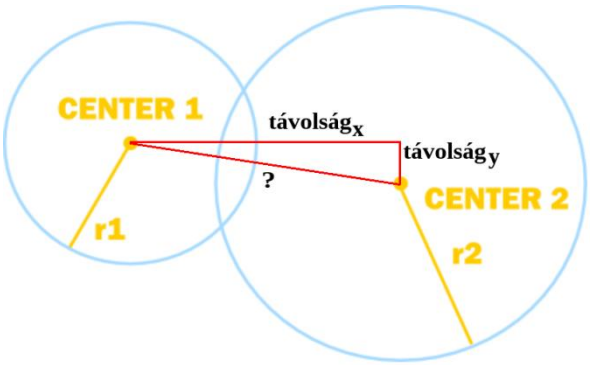


1. ábra. Befoglaló kör a játék objektumok körül. Látható, hogy a jobb oldali rakéta esetében már nem ideális a befoglaló kör alkalmazása

Hogy érdemes akkor megadni a befoglaló köröket? Általában kézzel úgy, hogy a különböző játék objektumokhoz kiegészítő leíró fájlt készítünk, amely nem csak az animáció fázisainak képi elemeit, hanem az azokhoz tartozó befoglaló köröket is tartalmazza.

3.2. Ütközések érzékelése

Az ütközések érzékelése ezen típusú befoglaló objektumok esetében a legegyszerűbb. Sokan, főleg kezdő játékfejlesztők ezért alkalmazzák előszeretettel. Mivel a két objektum szabályos, így akkor beszélhetünk ütközésről, amikor a befoglaló körök átfedik egymást, így ezt a tényt kell megvizsgálnunk. Két befoglaló kör pedig csakis akkor fedi át egymást, amikor a körök középpontjának távolsága kisebb mint a sugarak összege:



1. ábra. Ütközésvizsgálat befoglaló körökkel

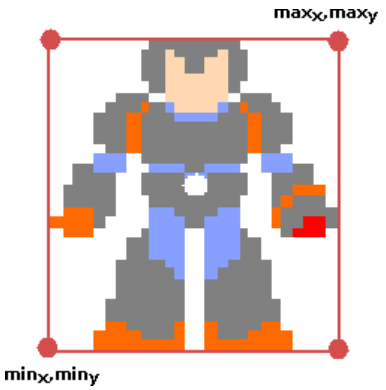
Az ütközés érzékelés pszeudó kódja:

```
bool checkCollision(center1, r1, center2, r2) {

        // Calculate the distance of the centers
        distanceX = center1.X – center2.X
        distanceY = center1.Y – center2.Y

        // Calculate distance based on Pythagorean theorem
        d = sqrt((distanceX * distanceX) + (distanceY * distanceY))

        // Check collision
        if (d <= (r1 + r2))
                return true; // Collision

        return false; // No collision
}
```

A megoldás legfőbb erénye - ahogyan a mintakódból is jól látszik - , hogy rendkívül egyszerű és számítási szempontból kis erőforrás igénnyel rendelkezik. Főként olyan esetekben javasolt, amikor olyan objektumaink vannak, amelyek jól illeszkednek a körbe.

4. Befoglaló doboz alapú ütközés érzékelés

Az ütközésvizsgálatok másik legegyszerűbb, de mégis legnépszerűbb megvalósítási formája a befoglaló doboz alapú megoldás (*rectangular collision detection*). Ebben az esetben az objektumot egy „dobozzal”, azaz egy négyzettel, vagy téglalappal vesszük körbe (Ábra 3). Népszerűsége egyrészt abból ered, hogy megvalósítása szintén egyszerű, másrészt adaptációs képessége révén viszonylag jól leírhatók vele az ütközésben résztvevő nem kör jellegű objektumok.



3. ábra. Legjobban illeszkedő befoglaló doboz. Láthatóan nincs üres pixel az alakzat legszélső pontjai és a doboz oldalai között

A befoglaló dobozt a legegyszerűbb esetben az objektum kétdimenziós képének, azaz a textúrájának a méretei alapján határozhatjuk meg. Ha a Sprite megfelelően van megrajzolva és nem tartalmaz felesleges átlátszó pixeleket a széleken, akkor a doboz a textúra betöltésekor könnyen kiszámítható. A doboz meghatározásánál általában arra törekednek, hogy a legjobban illeszkedő, úgynevezett "*Best fit rectangle*" vagy *"minimum bounding rectangle (MBR)*" modellt alkalmazzák. Ennek célja, hogy minimalizálják vagy elkerüljék a téves ütközés észleléseket. Például, ha az x tengely mentén megnövelnénk a doboz méretét, akkor az ütközést érzékelne akkor is, amikor az objektum még nem ért hozzá a falhoz.

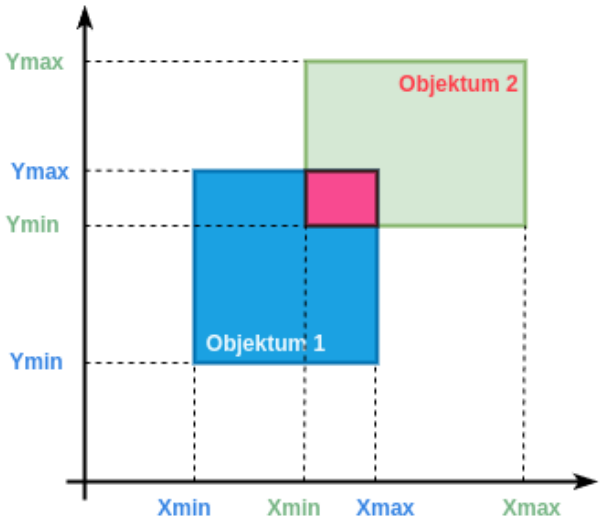
A következő kódrészlet egy lehetséges befoglaló dobozt megvalósító osztály leírást mutat be:

```
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
  bool mEnabled;                  // BB is enabled or not

public:
...
};
```

Két dimenzióban a befoglaló doboz négy sarkpontjának *(bbPoints[4]*) megadásával határozható meg, de a számítások gyorsítása érdekében célszerű a képernyő koordináta rendszeréhez viszonyított minimum (minpoint) és maximum (*maxpoint*) pontokat is tárolni. Az előző ábrán ezek a bal alsó és a jobb felső pontot jelentik. Ezen kívül a számításokhoz szükség lesz egy transzformációt végző mátrix osztályra, és gyorsítás céljából érdemes eltárolni a doboz szélességének és magasságának a felét is.

4.1. Doboz-doboz ütközés érzékelés

Az ütközés meghatározásának algoritmusa rendkívül egyszerű: ha két objektum befoglaló doboza átfedi egymást, akkor az objektumok ütköznek. Az alábbi ábra szemlélteti ezt a folyamatot:



4. ábra. AABB ütközés. Az átfedéssel az ütközés ténye megállapítható

Két doboz átfedésének számítása nem igényel nagy erőforrást. Azonban lehetnek olyan esetek, amikor akár több száz objektum van a képernyőn, így bármilyen optimalizáció csökkenti a CPU terheit. Ezért implementációs szempontból célszerű inkább azt az esetet vizsgálni, amikor nincs ütközés, mert ez egy gyorsabb számítást eredményez:

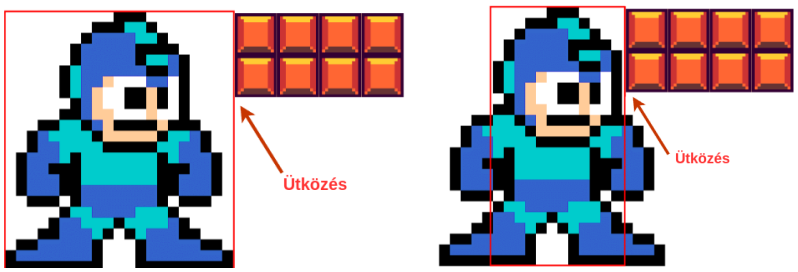
```
boolean CheckBoxOverLap(CBoundingBox2D box1, CboundingBox2D box2)
{
        if (box1.maxpoint.x < box2.minpoint.x ||
            box1.minpoint.x > box2.maxpoint.x){
           return false;
        }

        if (box1.maxpoint.y < box2.minpoint.y ||
           box1.minPoint.y > box2.maxpoint.y){
           return false;
        }
        return true;
}
```

Fontos megemlíteni a befoglaló doboz alapú megoldás egyik hátrányát is. Ha olyan objektumok ütköznek, amelyek „lyukasak” és csak a lyukas részeik érintkeznek, akkor nem mindig történik valódi ütközés. Bár a befoglaló doboz nem mindig követi pontosan az objektum formáját, hatékony és széles körben alkalmazott megoldás a gyakorlatban. Ennek oka az egyszerűsége, a kevesebb számítási igénye, és hogy sok játék esetében a gyors mozgás során nem észleljük, hogy az ütközés nem pontosan a pixel szintjén történik.

Bár fent úgy fogalmaztunk, hogy a célszerű a legszűkebb dobozt meghatározni, sok esetben ez sajnos nem elég a valósághű élmény eléréséhez. Tipikus példa a platform játékok, ahol a karaktert folyamatosan húzza a gravitáció, és addig nem esik le, amíg a doboza ütközik a talaj dobozával. Nem célszerű a dobozt ilyenkor a figura textúrájához illeszteni minden fázisban, különben előfordulhat, hogy az objektum nem esik le, amikor már le kellene. A következő kép egy klasszikus játékból készült, amelyben a karakter doboza pontosan a képének méretével egyezik meg lehetővé téve így azt, hogy akár 1 pixelen álljon a magasban.

5. ábra. Giana Sisters - 1987. A főhős képes volt stabilan állni a talajon, ha a befoglaló dobozának legszélső pixele már ütközött. A játékban egyébként ezt számos helyes ki kellett használni.

A hiba kiküszöbölésére kialakult egy nagyon egyszerű, de könnyen megvalósítható megoldás a gyakorlatban. A befoglaló doboz méretét nem pixelre pontosan az objektum képére számolják ki, hanem redukálják annak méretét valamilyen mértékben. Az alábbi ábra jól demonstrálja ezt:



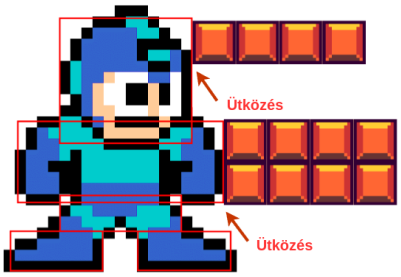
6. ábra. Redukált méretű befoglaló doboz valósághűbb ütközésérzékelést tesz lehetővé

A redukálás történhet valamilyen algoritmussal automatikusan, vagy akár kézi megadással. Ahogyan már a befoglaló körnél is említésre került, gyakran egy külön leíró fájlt alkalmaznak, amelyben lehetőség van minden doboz méretének kézi megadására.

4.2. Több befoglaló doboz alkalmazása

Amennyiben geometrialilag nem olyan jellegű szabályos objektumokat használunk a grafikus alkalmazásban, amelyek a téglalap segítségével megfelelően lefedhetők, úgy a normál AABB nem ad kielégítő megoldást. A széleken jelentkező üres területek gyakori fals ütközéshez vezethetnek, amelyek nagymértékben ronthatják a játékélményt.

A befoglaló dobozok egy továbbgondolt megoldása, ha nem egy, hanem akár több befoglaló objektumot is használunk az ütközési felület lefedésére. Az alábbi ábra erre mutat példát:



7. ábra. Több befoglaló doboz (*multiple bounding boxes*) használata nagymértékben javítja az ütközés érzékelésének hatékonyságát

A gyakorlatban számos esetben lehet hasznunkra ez a megközelítés, több szempontból lehet fontos. A legfőbb célja, hogy segítségével pontosíthatjuk az ütközésdetektálást. Vannak esetek, amikor egy doboz önmagában nem elegendő a kellő pontosságú ütközések megvalósításában, több doboz segítségével az eredmény javítható.

```
bool CheckCollision(CGameObject2D object1, CGameObject2D object2) {
    if (object1.isVisible() == false || object2.isVisible() == false)
        return false;

    vector <CBoundingBox2D> object1BBs = object1.getBoundingBoxes();
    vector <CBoundingBox2D> object2BBs = object2.getBoundingBoxes();

    for (int i = 0; i < object1BBs.size(); i++) {

        CBoundingBox2D object1Box = object1BBs[i];

        // Loop all the bounding boxes
        for (int j = 0; j < object2.size(); j++) {

           CBoundingBox2D object2Box =object2[j];
           bool result = CheckBoxOverLap(currentObjBox, objectCurrentObjBox);

            // Check overlapping case
            if (result == true) {
                return true;
            }
        }
    }

    return false;
}
```

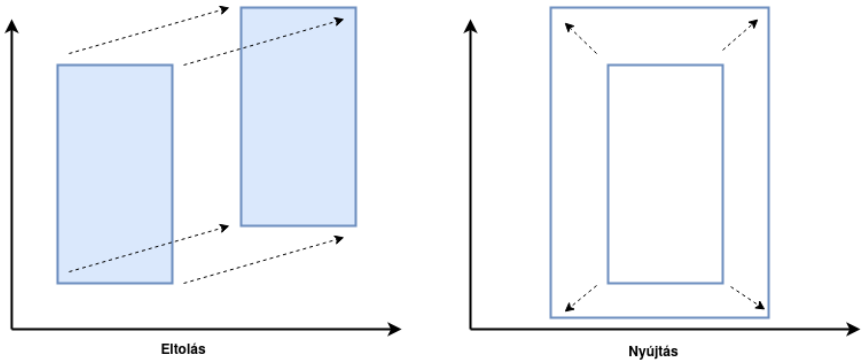
4.3. Befoglaló doboz mozgatása

Fontos néhány szót ejteni a befoglaló dobozok gyakorlati megvalósításáról. Az AABB alapú ütközésvizsgálat a gyakorlatban azt jelenti, hogy a játékobjektum mögött az AABB egyfajta “láthatatlan” dobozként jelent van, amelyet folyamatosan mozgatunk a játékobjektum elmozdulásának megfelelően. Gyakorlati szempontból bármilyen számítógépes játék készítése igényli az, hogy a befoglaló dobozokat meg is tudjuk jeleníteni a játék közben. Ennek számos praktikussági oka van. Legfőképp egyfajta debuggolási lehetőséget kínál a fejlesztők számára akkor, ha a játék meghajtó motorja, házon belül készül, nem pedig külső komponensként kerül beépítésre.

A dobozok mozgatása és a megjelenítés között azonban egy kisebb szakadék van. Míg a megjelenítést a GPU végzi a dobozok világ koordinátájának megfelelően és a pontos pozíció kiszámítása vertex árnyalóban történik, addig a dobozok bármilyen egyéb számítása (pl. átfedés vizsgálat) pedig a CPU oldalon. Az ütközésvizsgálat csakis akkor lesz pontos és működőképes, ha a doboz GPU általi kirajzolási a pozíciója egyezik a CPU oldalon végzett számításokkal.

Az objektum mozgatása (eltolás, forgatás, nyújtás) során tehát a doboz koordinátáját szintén transzformálni kell. A gyakorlati megvalósítás szempontjából érdemes megjegyezni, hogy egy objektum dobozának két változatát érdemes eltárolni: az eredeti és a transzformált változatot. Az eredeti példányt azért érdemes megtartani, mert bármilyen mozgás során ezt érdemes alapul venni, és a mozgási paramétereknek megfelelően ebből számolni a transzformált változatot. A transzformált változatot pedig azért célszerű tárolni, mert egy játék cikluson belül többször lehet szükség erre dobozra.

Az eltolás esetén nem jelentkezik komoly probléma, hiszen a doboz orientációját és méretét tartva mozog. Számítása egyszerű, az eredeti AABB koordinátáit transzformáljuk az új pozícióra. A nyújtási transzformáció esete is hasonló jelentkezik nagy probléma, hasonló az eltoláshoz annyi különbséggel, hogy itt már figyelembe kell venni azt, hogy mi a transzformáció középpontja. A játékot mozgató motor esetében fontos döntés az, hogy hol legyen az objektum lokális koordináta rendszerének origója [13]. Ez sok esetben az objektum közepe, vagy az objektum egyik sarokpontja. Ez a döntés befolyásolhatja a doboz nyújtásának számítását. Forgatás esetén azonban más jellegű nehézségekbe ütközünk.



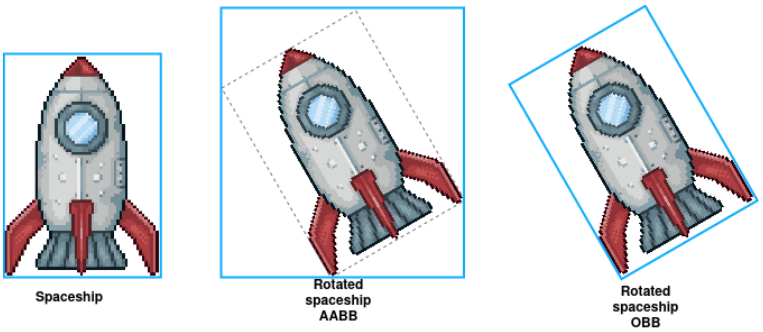
8. ábra. Befoglaló doboz eltolása és nyújtása

4.3.1. Befoglaló doboz forgatása

Az eddig bemutatottak alapján a dobozok forgatásnak természetes módjának tekinthetjük azt, ha a doboz sarokpontjait a megfelelő irányba elforgatjuk megkapva így az elfordult objektumot. A gyakorlatban azonban két megközelítés alakult ki a doboz orientációját tekintve:

- **Axis-Aligned Bounding Boxes (AABB):** olyan téglatest (2d-ben téglalap), amelynek minden éle egy koordinátatengellyel párhuzamos.
- **Oriented Bounding Box (OBB):** olyan téglatest (2d-ben téglalap), amely az objektum forgatásával együtt fordul.

A két megoldás közötti különbséget a következő ábrák szemléltetik:



9. ábra. AABB és OBB közötti különbség

Míg bárki számára az OBB tűnik a természetes megoldásnak, a gyakorlatban mégis inkább az AABB-t alkalmazzák. Ennek oka, hogy az OBB esetén két tetszőlegesen elforgatott doboz átfedését kell vizsgálni, amely matematikailag bonyolultabb algoritmussal írható le. Az AABB megvalósítása, a dobozok átfedésének kiszámítása (ütközésvizsgálat), a képernyő dobozával történő láthatósági vizsgálat (látszik-e a képernyőn vagy sem) lényegesen egyszerűbb, mint az OBB esetében.

Az elforgatott AABB hátránya azonban szemmel látható: a doboz forgatása során az eredeti mérete megváltozik, ezzel pedig az ütközés érzékelés hatékonysága nagymértékben romolhat.

A következőkben az AABB elforgatásának eljárását mutatjuk be. Megfogjuk figyelni azt, hogy a doboz mérete miért változik és hogyan kapunk az elforgatás után ismét a koordinátatengelyekkel párhuzamos dobozt.

4.3.2. AABB forgatása a gyakorlatban

Az AABB típusú befoglaló dobozok forgatása egy három lépéses folyamat:

1. Doboz négy pontjának transzformálása a forgatási szögnek megfelelően
2. Az elforgatott pontokból megkeressük a minimális és a maximális pontokat, tehát a új doboz határait
3. Ezen pontok alapján létrehozzuk a koordináta tengelyekkel párhuzamos új dobozt

A következőkben röviden bemutatjuk hogyan történik a gyakorlatban az AABB forgatása. A megoldás során arra van szükségünk, hogy a doboz 4 darab pontját elforgassuk, majd az elforgatott pontok alapján ismét meghatározzuk az AABB-t. A példa algoritmus a dobozt az x tengely mentén forgatja el, lényege röviden a következő:

```
// loop all the 4 points
for (unsigned int i = 0; i < AABB_POINTS; i++){
   // setup points as a vector
   CVector2 point(bbPoints[i].x,bbPoints[i].y);
   // rotate BB vector
   m_mTransformationMatrix.rotate_x(&point, angle);
   bbPoints[i].x = point.x;
   bbPoints[i].y = point.y;
}
```

Ezek után újra kell alkotni a befoglaló doboz, hogy ismét megfeleljen az AABB követelményeinek. Ezt két függvénnyel hajtjuk végre:

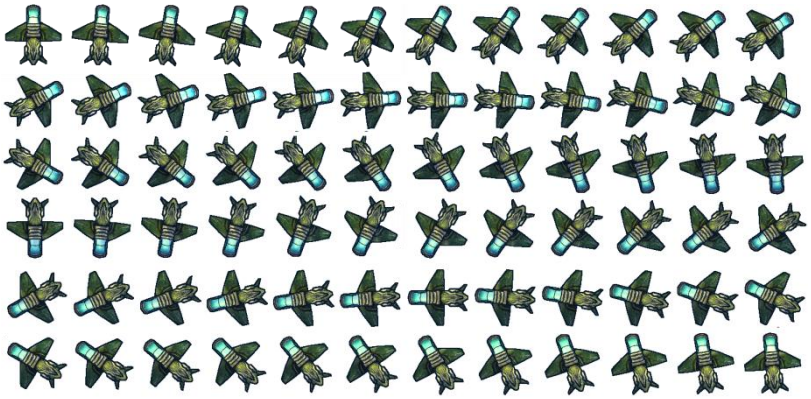
```
// Search min and max points
searchMinMax();

// setup AABB box
setUpBBPoints();
```

A *searchMinMax()* függvény megkeresi az elforgatott pontok közül az *x,y*

koordináták alapján a minimális és a maximális pontokat. Ezek után a *setUpBBPoints()* függvény pedig meghatározza a doboz 4 új pontját.

```
void setUpBBPoints()
{
    bbPoints[0].x = minpoint.x;
    bbPoints[0].y = minpoint.y;
    bbPoints[1].x = maxpoint.x;
    bbPoints[1].y = minpoint.y;
    bbPoints[2].x = maxpoint.x;
    bbPoints[2].y = maxpoint.y;
    bbPoints[3].x = minpoint.x;
    bbPoints[3].y = maxpoint.y;
}
```

Fontos megjegyezni, hogy a forgatás következménye a doboz méretének megváltozása. Az x ábrán szaggatott vonal jelzi az eredeti objektum "lefedett területét". Mivel a forgatás hatására megváltozott doboz mérete már sok esetben nem ideális az ütközések érzékeléséhez, ezért nem minden esetben alkalmazzák az ilyen jellegű forgatást. Talán a "hiba" kiküszöbölése lehet, ha az elforgatott doboz méretét valamilyen mértékben csökkentjük. Bizonyos játékokban azonban ettől eltérő, de egyszerű megoldást választanak. A játék objektum összes elfordított képét (akár 360 darabot) elkészítik egy rajzoló programmal és külön képként tárolják el. Így minden ilyen "fázisnak" saját befoglaló objektuma lehet, amely nem szenved a korábbi elforgatás során jelentkező doboz méret változástól. A játékmenet során pedig az aktuális elfordulási szögnek megfelelő képet jelenítik meg.

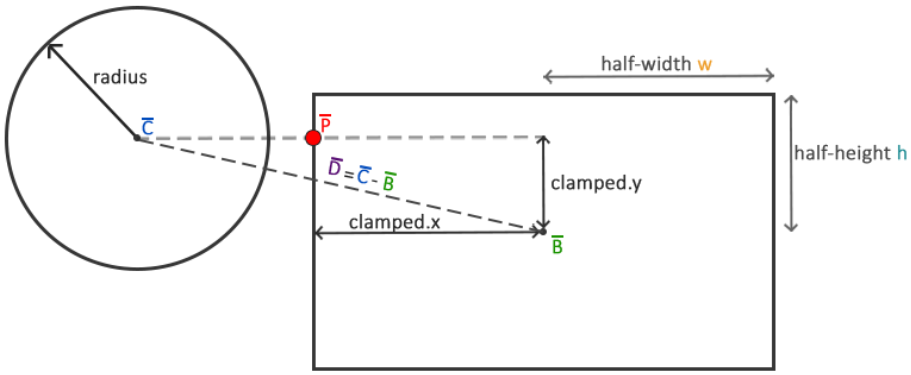


10. ábra A játékobjektum elforgatott fázisai

4.4. Befoglaló kör és doboz ütközésvizsgálat

Nem szabad elfeledkeznünk azonban egy a gyakorlatban is előforduló esetről, amikor különböző típusú befoglaló objektumot ütköznek össze. Nem kell messzire menni, egy tipikus falbontó (breakout) játékban a golyót egy befoglaló kör, a téglákat pedig befoglaló doboz határolja.

Az ütközések érzékelése ebben az esetben egy picit bonyolultabb lesz a korábbiakhoz képest. Az algoritmus logikája a következő [9]: Meg kell találni a befoglaló dobozon a körhöz eső legközelebbi pontot (*P*). Amennyiben a pont és a kör középpontjának távolsága kisebb, mint a kör sugara, úgy ütközés történt.



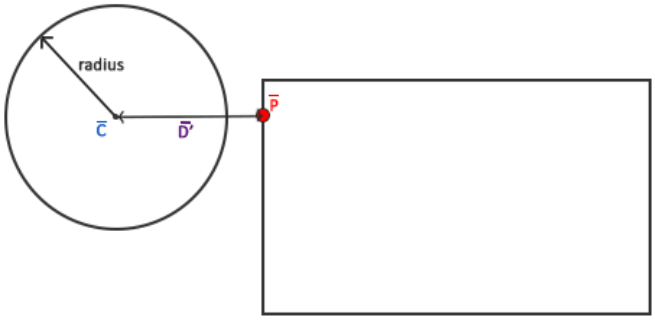
11. ábra. Kör - AABB ütközés geometriai szemléltetése

Az első lépés a kör középpontja (C) és a doboz középpontja (B) közötti távolságvektor (D) meghatározása. Majd ezt a vektort a doboz fele méreteinek megfelelően levágjuk (clamp). A doboz fele a doboz középpontja és a szélei közötti távolság. Az így kapott vágás utáni pont mindig a doboz egyik élén helyezkedik el valahol.

A vágás művelet azt jelenti, hogy egy adott értéket mindig egy értéktartomány halmazhoz igazítjuk. Az alábbiak szerint értelmezhetjük:

```
float clamp(float value, float min, float max) {
      return max(min,min(max, value));
}
```

A vágás után adódó *P* pont jelenti az AABB azt a pontját, amely legközelebb esik a körhöz. Az ütközés meghatározásához már csak annyit kell tennünk, hogy kiszámítjuk a kör középpontja (*C*) és a *P* közötti távolságot. Ehhez szükségünk van a *D* vektorra, amely a *C* és *P* vektorok különbségeként adódik. A *D* vektor hossza fogja adni a távolságot. Amennyiben ez az érték kisebb mint a kör sugara, úgy ütközés volt.



12. ábra. Kör - AABB ütközés geometriai szemléltetése

<u>Az ütközés érzékelés pszeudó kódja:</u>

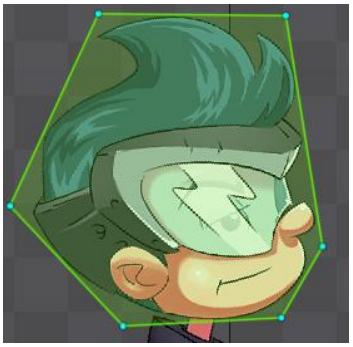
```
checkAABB_Circle_collision(AABB, Circle)

        // Distance of the two center
        Vector2 d_vector = Circle.center - AABB.center;
        Vector2 aabb_half_extents(AABB.sizeX / 2.0, AABB.sizeY / 2.0);
        float clampedX = clamp(d_vector.x, -aabb_half_extents.x,
        aabb_half_extents.x);
        float clampedY = clamp(d_vector.y, -aabb_half_extents.y,
        aabb_half_extents.y);
        Vector2 closestPoint = AABB.center + Vector2(clampedX,clampedY);
```

```
        Vector2 new_d_vector = closestPoint - Circle.center;

        // Calculate the distance
        diff = sqrt((new_d_vector.x * new_d_vector.x) + (new_d_vector.y *
        new_d_vector.y))

        if (diff <= (Circle.radius)
                return true; // Hit

        return false; // No hit
}
```

5. Befoglaló poligon

Természetesen a gyakorlatban dolgozhatunk olyan objektumokkal, amelyek esetében egyik eddig ismertetett megoldás sem nyújt megfelelő eredményt. Amennyiben még pontosabb ütközésérzékelésre van szükség, úgy a befoglaló poligon alapú megközelítést célszerű alkalmazni.

A befoglaló poligon egy tetszőleges pontból álló konvex vagy konkáv sokszög, amelynek célja, hogy a lehető legjobban illeszkedő formával vegyük körbe a játék objektumot. Az ütközésvizsgálatot pedig ezen poligonra hajtjuk végre.



13. ábra. Befoglaló poligon alkalmazása (Spine animation tool [11])

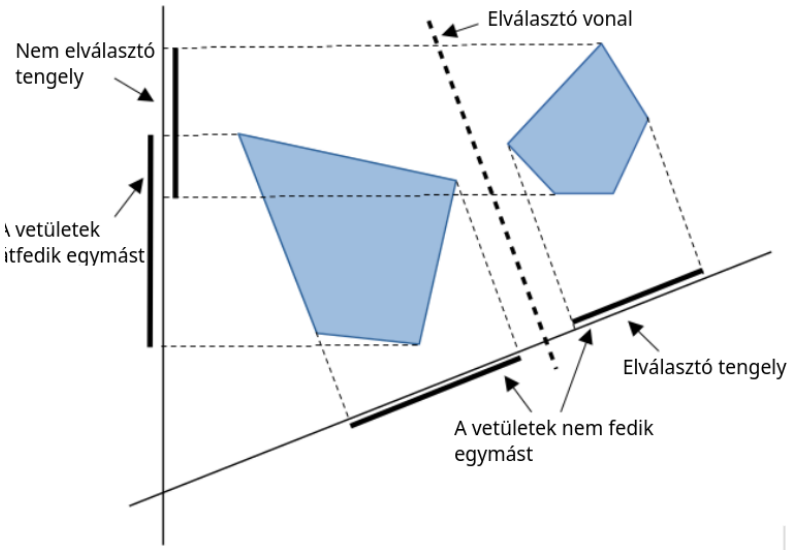
Láthatóan a poligon alapú megközelítés hatékonyan képes felvenni az objektum alakját. Napjainkban egyre több játékmotor kezdi már támogatni ezt a megközelítést. Általában külön szerkesztő felületet biztosítanak a poligon létrehozására és módosítására a játékmotor editorjában (pl. Unity [15], Cocos Creator [14]).

Két poligon ütközésének érzékelése már komplexebb feladat. Az irodalomban több megközelítés is elterjedt. Egyik ismert megoldás, amikor a poligont háromszögek halmazára bontjuk fel, majd ezen a halmazon végezzük el a szükséges típusú vizsgálatot. A vizsgálat típusa függhet az ütközésben résztvevő objektumok típusától. Amennyiben két poligon ütközik úgy poligon-poligon ütközésvizsgálatról beszélünk. Itt akár alkalmazhatunk háromszög-háromszög üzközés detektálást, vagy megvizsgálhatjuk, hogy az adott objektum poligonának valamelyik pontja benne van-e a másik objektum egy adott háromszögében. A következőkben egy gyakorlatban előszeretettel alkalmazott megoldást mutatunk be.

5.1. Elválasztó tengelyek elmélete

Az Elválasztó tengelyek elmélete (*Separating Axis Theorem*, *SAT*) egy olyan technika, amellyel megállapítható, hogy két konvex alakzat metszi-e egymást. Alkalmazható a minimális penetrációs vektor azonosítására is, amely hasznos a fizika szimulációjához és más alkalmazásokhoz. Ez a sokoldalú algoritmus kiküszöböli a specifikus ütközés észlelési kód szükségességét minden egyes alakzat típus-pár esetén, ami végső soron csökkenti a szükséges kód mennyiségét és megkönnyíti a karbantartási erőfeszítéseket.

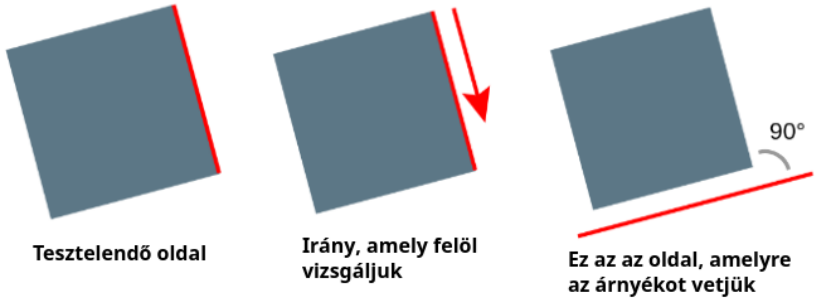
A SAT kimondja, hogy két konvex alakzat akkor és csak akkor nem metszi egymást, ha létezik legalább egy tengely, ahol az alakzatok ezen a tengelyen lévő merőleges vetületei nem metszik egymást.



14. ábra SAT működése

Tehát, ha két alakzat közé tudunk vonalat húzni anélkül, hogy egyiket sem érintenénk, akkor nincsennek átfedésben. Az objektumokat lerajzolva ez meglehetősen egyszerű. A gyakorlatban ezt úgy érhetjük el, hogy az alakzatokat egy tengelyre vetítjük, és ellenőrizzük az átfedést. Ha találunk egy tengelyt, ahol a vetületek nem fedik egymást, akkor azt mondhatjuk, hogy a két alakzat nem ütközik. Egy kétdimenziós objektum vetülete egydimenziós „árnyék”. Azt a vonalat, ahol az alakzatok vetületei (árnyékai) nem fedik egymást, elválasztási tengelynek nevezzük.

Programozási szempontból túl intenzív lenne minden lehetséges szöget ellenőrizni. A sokszögek természetéből adódóan csak néhány fontosabb kulcs szöget kell ellenőriznünk. A 2-dimenziós sokszögek kezelésekor csak azokat a tengelyeket kell megvizsgálni, amelyek merőlegesek az alakzat éleire. Ha a vetületek nem fedik egymást legalább az egyikben, az alakzatok nem fedik át egymást. A fenti képen ezek közül a tengelyek közül csak kettő látható, de az adott esetben 9 van belőlük (egy minden élhez).



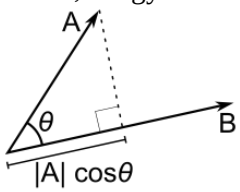
15. ábra. Objektum oldalának (élének) tesztelése

Az algoritmus logikája a következőképpen foglalható össze:

```
For each edge of both polygons:
  Find the perpendicular axis to the current edge.
    Loop through every point on the first polygon and project it onto the axis.
    (Keep track of the highest and lowest values found for this polygon).
    Do the same for the second polygon
    Check the projections for overlap.
    If the polygons don't intersect exit the loop
```

5.1.1. A vetítés

A vetítés a következő módon valósítható meg: tekintsünk minden élt A vektornak, a vetítési tengelyt pedig B vektornak, ahogy az alábbi ábrán [12] látható:



16. ábra. Objektum oldalának (élének) tesztelése

Két képlet segíthet ennek az ábrának az értelmezésében: a koszinusz trigonometrikus definíciója és a keresztszorzat geometriai definíciója:

$$cos\theta = \frac{|projection\ of\ A\ onto\ B|}{|A|} \tag{1}$$

$$A \cdot B = |A||B|cos\theta \tag{2}$$

Ezeket az egyenleteket kombinálhatjuk a vetítési képlethez:

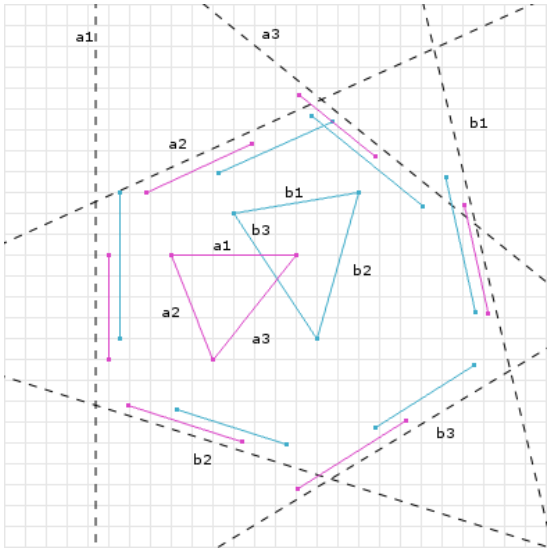
$$projection\ of\ A\ onto\ B = A \cdot \overline{B}, \text{ ahol } \overline{B} \text{ egy egységvektor a B irányába}$$

Így adott két vektor – egy a tengelyre (amelynek egységvektornak kell lennie), egy pedig az ütközési sokszögünk egyik sarkára – vetíthetjük a sarkot a tengelyre. Ha ezt minden sarok esetében megtesszük, akkor a sokszögből megtaláljuk a minimális és maximális vetületet [12].

Segítő módszer a min és max értékek megtalálásához:

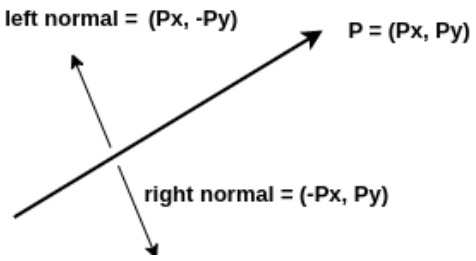
```
private MinMax FindMaxMinProjection(BoundingPolygon poly, Vector2 axis)
{
    var projection = Dot(poly.Corners[0], axis);
    var min, max = projection;

    for (var i = 1; i < poly.Corners.length; i++)
    {
        projection = Dot(poly.Corners[i], axis);
        max = max > projection ? max : projection;
        min = min < projection ? min : projection;
    }
    return new MinMax(min, max);
}
```

Ha mindkét alakzatra meghatározzuk a minimális és maximális vetületet, akkor láthatjuk, hogy átfedik-e egymást vagy sem.



16. ábra Két konvex alakzat metszéspont vizsgálata sok oldalellenőrzéssel [12]

Ha nincs átfedés, akkor találtunk egy elválasztó tengelyt, és leállíthatjuk a keresést. Geometriailag kimutatható, hogy a tesztelendő minimum egy a sokszög minden élnormáljával párhuzamos tengely – vagyis egy a sokszög élére merőleges tengely. Minden élnek két normálja van, egy bal és egy jobb:



17. ábra. Él normális

Az él normális az élvektorra merőleges egységvektor (egy vektor az él mentén) 2D-ben. Kiszámítható az x és y komponensek felcserélésével és az egyik tagadásával. Ezen normálok egyike kifelé, a másik pedig a sokszögben fog nézni, a pontok sorrendjétől függően (az óramutató járásával megegyezően / balra).

Mindkét irány működhet, amíg betartjuk az irányt. Tehát a normálértékeket úgy számítjuk ki, hogy a pontjainkon át iteráljuk, és létrehozunk vektorokat az egyes élekre, majd kiszámítjuk az adott élre merőleges vektort.

```
public struct BoundingPolygon {
       Vector2[] mCorners;
       Vector2 mCenter;
       Vector2[] mNormals;

  public BoundingPolygon(Vector2 _center, Vector2 _corners) {
        mCenter = _center;
        mCorners = _corners;
        var normals = new HashSet<Vector2>();

        var edge = Corners[mCorners.length - 1] - mCorners[0];
        var perp = new Vector2(edge.Y, -edge.X);
        perp.Normalize();
        normals.Add(perp);

        for (var i = 1; i < mCorners.length; i++) {
            edge = mCorners[i] - mCorners[i - 1];
            perp = new Vector2(edge.Y, -edge.X);
            perp.Normalize();
            normals.Add(perp);
        }
        mNormals = normals.ToArray();
    }
}
```

Két *BoundingPolygons* közötti ütközés észleléséhez végig iterálunk a kombinált normálisaikon, mindegyikhez generálunk egy *MinMax* értékét, és teszteljük az átfedést. Példa kód:

```
public bool Collides(BoundingPolygon p1, BoundingPolygon p2) {
    foreach(var normal in p1.mNormals){
        var mm1 = FindMaxMinProjection(p1, normal);
        var mm2 = FindMaxMinProjection(p2, normal);
        if (mm1.Max < mm2.Min || mm2.Max < mm1.Min) return false;
    }
    foreach (var normal in p2.mNormals) {
        var mm1 = FindMaxMinProjection(p1, normal);
        var mm2 = FindMaxMinProjection(p2, normal);
        if (mm1.Max < mm2.Min || mm2.Max < mm1.Min) return false;
    }
    return true;
}
```

A SAT számos tengelyt tesztelhet átfedés szempontjából, azonban az első tengelynél, ahol a vetületek nem fedik egymást, az algoritmus azonnal kiléphet, megállapítva, hogy az alakzatok nem metszik egymást. Ennek a korai kilépésnek köszönhetően a SAT ideális olyan alkalmazásokhoz, amelyekben sok objektum van, de kevés ütközés (játékok, szimulációk stb.).

6. Összefoglalás

Az ütközések érzékelése szerves részét képezi a mai számítógépes vizualizációnak legyen az egy játék vagy bármilyen alkalmazás, ahol interakcióra van szükség. Bár a mai szoftverek egyre komplexebbek, egyre több mozgó objektummal dolgoznak, de a háttérben megvalósított ütközések érzékelése jól ismert szakmai

megoldásokkal történik. Bár a hardverek teljesítménye sokat nőtt az utóbbi években, a pixel szintű ütközésérzékelés továbbra sem kínál hatékony alternatívát nagy számításigénye miatt, helyette a befoglaló objektumon alapuló algoritmusok dominálnak. Ezek a megoldások rendelkeznek a fals ütközés érzékelés hibájával. Ezért egy modern grafikus motor általában több megoldást is implementál annak érdekében, hogy lehetőséget kínáljon a módszerek valamilyen szintű kombinációjára a megfelelő eredmény elérése érdekében. Jelen cikk a legfontosabb megközelítéseket tekintette át gyakorlati szempontból. A bemutatott módszerek előnyeit és hátrányait vizsgálva lehetőséget adnak egy hatékony ütközési rendszer megtervezéséhez és elkészítéséhez.

Irodalom

[1] Charles Kelly: Programming 2D Games, A K Peters/CRC Press; 1st edition, 2012.
[2] Jason Gregory: Game Engine Architecture, A K Peters/CRC Press; 3rd edition, 2018.
[3] Tomas Akenine-Moller, Eric Haines, Naty Hoffman: Real-Time Rendering, 3rd Edition, A K Peters/CRC Press; 2008.
[4] Lazaridis, L., Papatsimouli, M., Kollias, KF., Sarigiannidis, P., Fragulis, G.F.: Hitboxes: A Survey About Collision Detection in Video Games. In: Fang, X. (eds) HCI in Games: Experience Design and Game Mechanics. HCII 2021. Lecture Notes in Computer Science(), vol 12789. Springer, Cham. https://doi.org/10.1007/978-3-030-77277-2_24, 2021.
[5] K. Guo and J. Xia: An Improved Algorithm of Collision Detection in 2D Grapple Games, 2010 Third International Symposium on Intelligent Information Technology and Security Informatics, Jian, China, pp. 328-331, 2010, doi: 10.1109/IITSI.2010.176.
[6] Benjamin Rodrigue: Algorithmic and Architectural Gaming Design: Implementation and Development, Chapter 10: Collision Detection in Video Games, DOI: 10.4018/978-1-4666-1634-9.ch010, 2012.
[7] Thomas Schwarzl: 2D Game Collision Detection: An introduction to clashing geometry in games, Createspace Independent Publishing Platform, 2012.
[8] Intro to Collision Detection: Collision Detection Basics, http://www.kilobolt.com/collision-detection-basics, 2024.
[9] Learn OpenGL - Collision detection: http://learnopengl.com/#!In-Practice/2D-Game/Collisions/Collision-detection, 2024
[10] Nathan Bean, Foundations of Game Programming, CIS 580 Textbook, Kansas State University, 2021.
[11] EsotericsSoftware: Spine - 2D animation tool, http://esotericsoftware.com/, 2024
[12] SAT (Separating Axis Theorem), https://dyn4j.org/2010/01/sat/, 2024
[13] Peter Mileff: Practical Guide To Implement a Simple 2D Game Engine, Production Systems and Information Engineering, Volume 11 (3), pp 27-50, doi: 10.3.2968/psaie.2023.3.3, 2024.
[14] Cocos game engine, https://www.cocos.com, 2024.
[15] Unity Engine, https://unity.com, 2024.