Production Systems and Information Engineering
Volume 12 (4), pp. 1-21
doi: https://doi.org/10.32968/psaie.2024.4.1

2D JÁTÉKVILÁGOT MEGVALÓSÍTÓ KOMPONENSEK TERVEZÉSE ÉS MEGVALÓSÍTÁSA

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

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

Abstract. A számítógépes vizualizáció ma már egy nagyon fontos területet képvisel az emberek életében. Gyakorlatilag szinte minden területen megjelent, ma már elengedhetetlen. A területen belül a játékfejlesztés hatalmas iparággá nőtte magát ki. Egy színvonalas 2D vagy 3D játék elkészítése ma már rendkívül komoly fejlesztői munkát igényel. Bár az évek során számos platform, developer API és játék motor alakult ki, amelyek nagymértékben segítik a fejlesztők munkáját, sokan vannak, akik szeretnék megérteni a játékok mögötti technológia működését. Ezért maguk építik fel a különböző szoftver komponenseket, melyekből komplex rendszert építenek és írják meg a kép szintézishez és egyéb területhez szükséges algoritmusokat. Jelen publikáció erre a területre fókuszál. Célja, hogy bemutassa azokat az alapvető szükséges komponenseket, amelyek szükségesek egy két dimenziós játék megvalósításához, valamint jó alapul szolgálhatnak további magasabb szintű struktúrák kialakításához.

Kulcsszavak: játékfejlesztés, virtuális világ, játék komponensek

1. Bevezetés

A kétdimenziós megjelenítés fontos szerepet tölt be a mai modern három dimenziós leképzés mellett. Azt lehet mondani, hogy bár a világ a megjelenítés terén 3D irányába mozdult el az utóbbi évek során, a kétdimenziós megoldások kiegészítő technikaként mindig jelen voltak és lesznek is a későbbiekben. A számítógépes alkalmazások több rétege tartozik ide. Bármilyen olyan szoftver, amely rendelkezik valamilyen grafikus menü- vagy ablakozó rendszerrel, továbbá a kétdimenziós számítógépes játékok nagy képviselői a területnek. A menürendszer szempontjából mindig vannak próbálkozások arra, hogy ezeket a részrendszereket is a háromdimenziós térben próbálják szemléletesebbé tenni, bár működnek, de többnyire ezek a megoldások visszatérnek a 2D leképzéshez. Ezeknél a rendszereknél a sebesség a mai rendszerek teljesítményéhez viszonyítva nem kritikus. Leginkább statikus képek váltakoznak, melyek száma kevés, egyéb transzformációk (forgatás, nyújtás, döntés) azonban nem nagyon jellemzőek. A számítógépes játékoknál azonban pont az ellenkezője tapasztalható. Ilyen szoftverek esetén általában nagy mennyiségű folyamatosan változó objektumok halmazát kell raszterizálni, amely jelentős erőforrást igényel dinamikusan változva

a mozgatott elemek függvényében. A mai rendszerek tipikus jellemzője a nagy textúrahalmaz, animációk és transzformációk dinamikus alkalmazása a magasabb felhasználói élmény elérése érdekében. Az igényeknek megfelelően a képernyőfelbontás nagy, minőségi textúrák használata már elengedhetetlen, amely tovább növeli a követelményeket a teljesítménnyel szemben.

A játékipar népszerűsége folyamatosan nőtt az utóbbi 20 évben és ez a folyamat továbbra is töretlen. Ennek egyik mozgatója az ugrásszerű technológiai fejlődés. Ma egy középkategóriás mobil eszköz/tablet már jobb teljesítményre képes, mint mondjuk egy régi Pentium 4, vagy egy Dual Core Pentium. Mindazonáltal ezek az eszközök sokkal több emberhez jutnak el, mint korábban. A technológia ugrás szerű fejlődését a játékok is követik. Mind látványvilágban, mind komplexitásban és játékélményben is jól látszik a fejlődés. Egyre magasabbak az elvárások, ehhez pedig jó alapot nyújtanak a fejlődő hardverek. Több platformon is megjelent a játékfejlesztés a modern vizualizáció [16]. Például a HTML5 alapú megoldások. Ennek következménye, hogy az utóbbi években megnövekedett az aktív (kortalan) játékosok száma.

A játékok fejlesztésére több lehetőség adódik:

Fejlesztés játék editorral: ma már rendelkezésre állnak olyan szoftverek, amelyek lényegében ténylegesen betöltik a játék editor (Roblox [10], Unity [5], Game Editor, Construct 3) szerepét. Itt a legfontosabb cél, hogy olyan szintű szerkesztő szoftvert dolgozzanak ki, amellyel vizuálisan össze lehet rakni a játékot. Természetesen meghagyják a lehetőséget arra, hogy ha valami egyedi, az editor által közvetlenül nem támogatott funkciót szeretnénk megvalósítani, akkor ez megoldható legyen. Ezekre általában azt a megoldást alkalmazzák, hogy a kérdéses komponensekhez, objektumokhoz egy valamilyen programozási nyelven (pl.: Javascript, C#, stb) írt kódot, szkriptet tudunk készíteni. Bár ez a fajta megközelítés kialakulása a fejlődés természetes velejárója, sok programozó számára idegen az a megközelítés, amikor nem látjuk a teljes kód bázist egyben, kicsit elveszti a varázsát a klasszikus játékfejlesztők körében.

Fejlesztés játék motorral: játékmotorok már szinte a számítógépes vizualizáció PC-s korszakának kezdetétől fogva léteztek. Feladatuk, hogy a játékfejlesztés folyamatát nagymértékben segítsék azzal, hogy számos dologra kész, keretrendszer szintű megoldást kínálnak [1]. Ma már számos professzionális játékmotor érhető el, melyek rengeteg tudást és munkát integrálnak magukba. Nyugodtan kijelenthetjük, hogy a legtöbbjük magas színvonalú megjelenítést és optimalizált teljesítményt nyújt. Éppen ezért nem véletlen, hogy számos motor nem férhető hozzá ingyen. Egy igazán jó játék motor ára akár nagyon magas is lehet.

A játékmotorok API szinten nyújtanak támogatást. Ezeket a könyvtárakat a programhoz linkelve érhetjük el a funkciókat. További jellemzőjük - főként a háromdimenziós motoroknál -, hogy rendelkeznek valamilyen jellegű szerkesztővel (2D vagy 3D). Ezek a felületek nem a játék editor szerepét töltik be, inkább kiegészítő tool-ok a pályák szerkesztéséhez, az anyagtulajdonságok beállításához, vagy akár egyéb dolgokhoz.

Fejlesztés natívan: az utolsó kategóriába pedig azon fejlesztések tartoznak, ahol nem használunk kimondottan valamilyen külső játékmotort, hanem a fejlesztők mindent maguk készítenek el házon belül. Természetesen ez igényli a leginkább komplex tudást és fejlesztést. Cserébe azonban olyan tudást szereznek meg a fejlesztők, amellyel a megjelenítést részleteiben fogják érteni. Az ilyen jellegű fejlesztéseknél idővel saját játékmotor is gyakran kialakul, amely már

szoftvertervezési és fejlesztési szempontból ad magas fokú tapasztalatot.

A kétdimenziós számítógépes játékvilág különböző elemekből épül fel. A gyakorlatban, implementációs és tervezési szinten ezeknek számos megvalósítása lehet, de alapvető követelmény, hogy a világ elemei valamilyen újrahasználható elemekre támaszkodva legyenek kivitelezhetők [2]. Az ilyen jellegű megvalósítás teszi majd lehetővé azt a programozók számára, hogy hatékony és dinamikus fejlesztés segítségével produktívan legyenek képesek haladni a játék fejlesztésével.

Jelen cikkben a játékvilág felépítését vizsgáljuk meg részleteiben. Azt, hogy melyek azok a fontos elemek, amelyre mindenféleképpen szükség van és melyekre támaszkodni lehet egy bonyolultabb játék, vagy akár játékmotor készítésekor. Bemutatunk egy olyan javasolt tervezési struktúrát, amely működőképes, hatékony és jó alapot kínálhat a jövőben egy-egy komplexebb játékmotor számára. A részletezett struktúra a natív fejlesztést veszi alapul, az OpenGL vagy DirectX-en kívül nincs további olyan API réteg, amely extra funkciókkal támogatná a megjelenítést vagy a játék készítését. Továbbá a cikk kizárólag a megjelenítéshez szükséges magasabb szintű struktúrákkal foglalkozik, feltételezi, hogy azokat az alacsonyabb szintű építő köveket (2D textúra kirajzoló árnyaló megléte, mátrixok, képek betöltése, input kezelés), amelyek szükségesek a működéshez és a vizualizációhoz.

2. Irodalmi áttekintés

A képszintézis az első számítógépek megjelenénekor még szoftveres úton történt. Nem voltak még játékmotorok, keretrendszerek, a fejlesztők maguk készítettek el mindent. A korai évek során született raszterizálók közül legkiemelkedőbb eredmény az ID software által készített Quake I, II szoftveres renderer (1996), amely az első valós háromdimenziós motor volt [6]. Grafikus megjelenítő alrendszereit Michael Abrash koordinációjával készült jellegzetesen a Pentium processzorcsaládra optimalizálva kihasználva a nagyszerű MMX utasításkészlet. A később született eredmények közül főleg az Unreal motort (1998) lehet kiemelni, amely nagyon gazdag funkcionalitással rendelkezett. A GPU renderelés folyamatos térnyerése után a szoftveres megjelenítés egyre inkább háttérbe szorult. Az idő előrehaladtával és az internet terjedésével elérhetővé váltak az első játékmotorok, melyek ma már a játékfejlesztés területén kulcsfontosságú szerepet játszanak. Ezek a motorok biztosítják a fejlesztők számára a szükséges eszközöket a 2D-s eszközök létrehozásához, kezeléséhez és rendereléséhez, lehetővé téve a látványos és funkcionális játékok létrehozását. Az alábbiakban áttekintjük a legismertebb játékmotorokat:

Unity

A Unity az egyik legsokoldalúbb játékmotorként ismert, amely elsősorban erős 3D képességeiről híres, de a 2D játékfejlesztésben is rendkívül hatékony. A Unity 2D eszközkészlete olyan funkciókat tartalmaz, mint a Tilemap rendszer, Sprite-kezelés és kifejezetten 2D játékokhoz igazított fizika, ami nélkülözhetetlen eszközzé teszi a fejlesztők számára. A motor többplatformos képességei, valamint átfogó asset store-ja lehetővé teszik a fejlesztők számára, hogy hozzáférjenek egy széles körű előre elkészített 2D eszköztárhoz és pluginokhoz, jelentősen felgyorsítva a fejlesztési folyamatot [9].

Unreal Engine

Az Epic Games által fejlesztett Unreal Engine egy másik erőmű a játékiparban. Bár hagyományosan a 3D képességeiről ismert, az Unreal Engine egyre inkább

támogatja a 2D játékfejlesztést is, különösen a Paper2D eszközkészleten keresztül. A Paper2D átfogó készletet biztosít a 2D sprite-ok, animációk és szintek létrehozására és kezelésére. Az Unreal Engine erőteljes renderelő motorja lehetővé teszi a magas minőségű 2D grafika előállítását, még a bonyolult jelenetekben is [15].

Godot Engine

A Godot egy nyílt forráskódú játékmotor, amely nagy népszerűségre tett szert könnyű, rugalmas és intuitív kialakítása miatt. 2D motorja teljesen elkülönül a 3D motorjától, biztosítva, hogy az eszközök és a teljesítmény optimalizálva legyenek a 2D játékfejlesztéshez. A Godot jelenetrendszere lehetővé teszi a fejlesztők számára az újrahasznosítható elemek létrehozását, míg a Pythonhoz hasonló GDScript egy egyszerű szkriptnyelvi környezetet kínál [14].

Cocos2d

A Cocos2d egy nyílt forráskódú keretrendszer, amely jelentős hatással volt a 2D játékok fejlesztésére, különösen a mobil platformokon. A legnépszerűbb változata, a Cocos2d-x, egy többplatformos keretrendszer, amely C++ nyelven íródott, és támogatja a Lua és JavaScript nyelveket is. Robusztus eszközöket biztosít a sprite-kezeléshez, a fizikához és a felhasználói felület tervezéséhez, így különösen alkalmas a mobil játékfejlesztésre [13].

Phaser

A Phaser egy gyors, nyílt forráskódú HTML5 játékkeretrendszer, amely különösen alkalmas olyan 2D játékok létrehozására, amelyek böngészőkben futnak. A JavaScript-re épülő Phaser átfogó funkciókészletet kínál, beleértve a rugalmas plugin rendszert, a robusztus sprite-kezelést és a fejlett fizikai motorokat, mint például az Arcade Physics és a Matter.js [12].

Construct

A Construct egy játékmotor, amelyet kifejezetten 2D játékok létrehozására terveztek anélkül, hogy hagyományos programozásra lenne szükség. Vizuális szkriptelési felülete az egyik meghatározó jellemzője, amely lehetővé teszi a nem programozók számára, hogy összetett játékokat fejlesszenek egy eseményalapú rendszer segítségével. A Construct különösen népszerű az oktatási környezetekben és a hobbifejlesztők körében, mivel könnyen hozzáférhető és használható [11].

GameMaker

A GameMaker egy jól ismert játékmotor, amelyet kifejezetten 2D játékok fejlesztésére terveztek. A GameMaker Studio 2, a motor legújabb verziója, egy erőteljes, ugyanakkor felhasználóbarát eszközkészletet biztosít, amely lehetővé teszi a fejlesztők számára, hogy gyorsan és hatékonyan készítsenek 2D játékokat. A GameMaker nyújt kód nélküli fejlesztési lehetőségeket is a Drag-and-Drop felületével, de lehetővé teszi a fejlettebb programozást is a GameMaker Language (GML) használatával. A GameMaker különösen népszerű az indie fejlesztők körében, és számos sikeres 2D játék, például a "Undertale" is ezen a platformon készült [8].

A játékfejlesztéshez választott játékmotor gyakran a projekt specifikus követelményeitől függ, például a célplatformtól, a kívánt grafikai hűségtől és a fejlesztő ismereteitől a motor eszközeivel kapcsolatban [3]. A Unity és az Unreal

Engine kiterjedt funkcióik és többplatformos képességeik miatt előnyösek, míg a Godot és a Cocos2d nyílt forráskódú jellegük és a 2D fejlesztéshez való optimalizálásuk miatt kedveltek. A Phaser és a Construct kiemelkedik a használhatóságuk és a specifikus platformokra, például a webböngészőkre és az oktatási környezetekre való fókuszuk miatt.

3. A virtuális világ elemi egységei

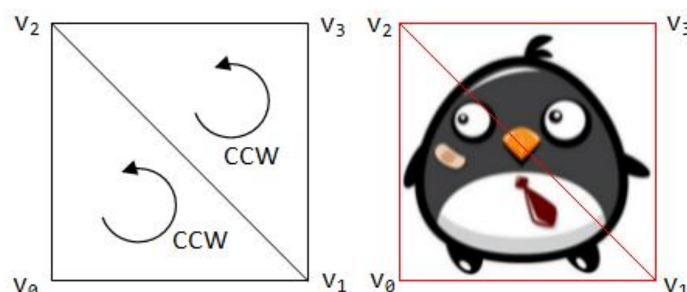
Aki foglalkozott már számítógépes játékfejlesztéssel, az tudja, hogy a grafikus alkalmazások számára az első mérföldkő minden esetben az, hogy tudjunk egyáltalán valamit kirajzolni a képernyőre. (Feltéve, hogy natív fejlesztésről beszélünk). Ma már sok esetben ez sem triviális, mert a számítógépes vizualizáció belépési szintje jelentősen megemelkedett. Ma már egy API szintű "Hello World" alkalmazást készítése sem egyszerű, hiszen geometriával, mátrixokkal, shaderekkel és további egyéb feladatokkal kell foglalkozni egy primitív demó esetében is, és ez alól a kétdimenziós megjelenítés sem kivétel.
A következőkben bemutatjuk a 2D motorok grafikus megjelenítés alapjául szolgáló jellemző általános elemi egységeit, azok kapcsolódási jellegeit, amely alapjául szolgálhat egy két-dimenziós játék motor fejlesztéséhez. A bemutatott mintakódok C++ nyelven kerültek megvalósításra.

3.1. Egyszerű textúra objektum

A grafikus API-k úgynevezett textúrákkal dolgoznak. Számunkra jelentsen ez most egy kétdimenziós (általában alfás) képet. A képet a használat előtt be kell tölteni a központi memóriába, majd ezután pedig a GPU memóriába. Természetesen a GPU memória mérete limitált, így nem tölthetünk be oda általában mindent. A legtöbb esetben ezért van az, hogy a játékok a különböző pályákhoz tartozó grafikai adatokat csak a használatkor töltik be GPU memóriába. Amennyiben sok grafikai adatot szeretnénk eltárolni a GPU memóriában a textúra tömörítés lehetősége adódik vagy a GPU Level streaming.

Egy grafikus alkalmazás számos pontján igényel statikus elemeket. Ilyen például egy háttérkép, egy mozgó felhő, de akár a menü gombok is. A játékmotor egyik alap eleme tehát egy olyan statikus objektum, amelyre a későbbiekben további komplexebb elemek építhetnek. Egy egyszerű statikus objektum, amennyiben a poligon alapú raszterizációt alkalmazzuk általában két háromszögből épül fel.

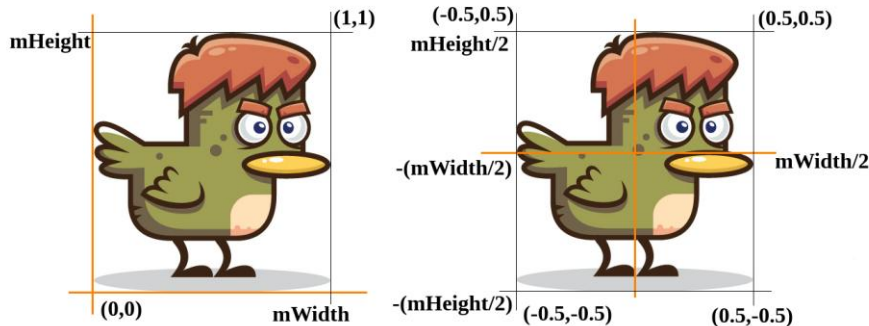
A GPU-ban való tárolás miatt szükség van egy API specifikus részre, amely kezeli ezt. OpenGL esetén ez a Vertex Array Object.



Ábra 1. Kétdimenziós képek gyakori háromszög alapú képzési formája

A vertex adatok megadásánál már rögtön két megközelítés alakult ki attól függően, hogy mi legyen a sprite rögzítési pontja (anchor point). Az alábbi ábra a két

leggyakrabban alkalmazott formát mutatja be.



Ábra 2. Népszerű koordináta-rendszerek

Mindkét megoldásnak megvannak a maga előnyei és hátrányai. Az elhelyezés a kirajzolás logikáját befolyásolja. Az első esetben ha kitesszük az objektumot egy koordinátára, akkor a kirajzolás attól a ponttól kezdve jobbra és felfelé fog történni, ahogyan a rajz koordináta-rendszerének origója jelöli. A második esetben pedig éppen az objektum közepe lesz a megadott pozíció. A rögzítési pont ezen felül befolyásolja az objektum forgatását is, mégpedig úgy, hogy a rögzítési pont lesz a forgásközéppont. Természetesen ha a baloldali megoldást választottuk, attól még forgatás esetén a forgásközéppontot külön megtudjuk adni.

```
class CTexture {

    CVector2 mPosition;          // position of the texture
    CVector2 mRotation;          // orientation of the texture
    CVector2 mScale;             // scale of the texture
    bool bVisible;               // Texture is visible or not
    float m_WidthOriginal;       //  Stores the original width of the texture
    float m_HeightOriginal;      // Stores the original height of the texture
    CVAOobject* mTextureVAO;     //  Holds  texture  Vertex,  Texture  and  Color
information
    unsigned int mID;            // Global ID of the texture
    sColor sColor;               // Color information
    string mFilename;            // Holds the filename of the texture
    string mName;                // Material name
    float mWidth;                // Stores the width of the texture
    float mHeight;               // Stores the height of the texture
    unsigned int mTextureID;     // Holds the texture ID


public:

    CTexture();

    CTexture& operator= (const CTexture& _texture);   // overloaded operator
    Draw(...);                                       // Draw texture

 [...]
};
```

A felépített osztály tartalmazza a két dimenziós megjelenítéshez szükséges legfontosabb alap tulajdonságokkal. Éppúgy alkalmas lesz egy játékobjektum képének a megjelenítésére, mint egy menü elemre is. Az alapelemek mellet célszerű helyet kapnia egy „másoló függvénynek” is. A minta C++ jellege miatt, itt ezt a szerepet az egyenlőség operátor túlterhelésére bízzuk, amely azért szükséges, ha egy *CTexture* osztály adatait képesek legyünk egy sor hívással átadni egy másik osztálynak. Egy konkrét példa a arcade játékokban megjelenő lövés folyamat. Amikor az objektumok lövedéket lőnek ki, akkor létrehozunk egy új lövedék osztály példányt (esetleg a lövedékek egy előre létrehozott pooljából választunk). Ezt az új lövedéket nagyon könnyen feltölthetjük adatokkal, amennyiben egy

meglévőt (az anya objektumot) másolunk le. A copy konstruktor/operátor átdefiniálás nélkül az adatokat egyesével kellene átadni az új osztálynak, amely hosszú távon rontja a kód minőségét.

3.2. Ütközés detektálás

A játékprogramok elengedhetetlen eleme az objektumok egymással való interakciója, azaz annak a vizsgálata, hogy két objektum mikor ütközik egymásnak, mikor érintkeznek. Ez valójában nem csak a játékok világára jellemző, hanem ugyanezen elveket alkalmazzuk akkor is, amikor például az egeret egy menüelem felé helyezzük. Természetesen a számítógépes játékokban az interakciók megfelelő szintű érzékelésének domináns szerepe van, hiszen a játékélmény ezen interakciók hatására alakul ki. Például egy akciójátékban játékban a lövedék eltalálja az ellenséget, vagy egyszerűen csak ne tudjon a főhős átsétálni a labirintus falán.

Az ütközésvizsgálat lényege nagyon leegyszerűsítve az, hogy valahogyan algoritmikusan érzékelni kell, hogy két vagy több objektum kétdimenziós képe átfedi-e egymást. Amennyiben pontosabban fogalmazunk, a probléma kicsit bonyolultabb ennél: azt jelenti, hogy egy objektumnak van-e olyan pixele, amely átfedi egy másik objektum pixelét.

Egy játék fejlesztése során minden bizonnyal eljön az a fontos pillanat, amikor döntenünk kell arról, hogy milyen ütközésdetektáló rendszert, milyen modellt alkalmazzunk. A döntés nem mindig könnyű és egyértelmű. Vannak olyan típusú játékok, ahol az interakciók nagyon komplexek lehetnek, valamint sokszor nem is látszik minden probléma előre. Ennek ellenére az alkalmazott modell fontos, hiszen nagymértékben hatással van a fejlesztési időre és magára a játékélményre is. Alapvetően az alábbi két csoportba sorolhatjuk az ütközésdetektáló rendszereket:

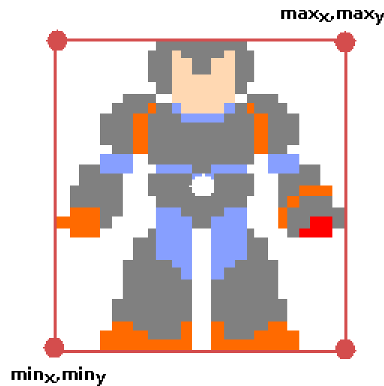
- **Pixel alapú ütközésvizsgálat:** az ütköző objektumokhoz tartozó képek pixeleinek átfedését vizsgálja. Precíz, valós ütközést képes érzékelni

- **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 komplikált lehet, attól függően, hogy milyen bonyolult az objektumhoz tartozó textúra. Emiatt a játékfejlesztők ahol tudnak inkább valamilyen objektum(ok)ba próbálják meg befoglalni a mozgatott elemeket és erre elvégezni az ütközések vizsgálatát. Általában a választás a kör, vagy dobozra szokott esni, mert nagyon egyszerű elemekről van szó. A velük való későbbi számolások (ütközésvizsgálat, forgatás, eltolás, stb) közel sem annyira számításigényesek, mint például egy befoglaló poligon, vagy a pixel szintű vizsgálat esetében. Bár nem közelítik jól az objektumot, mégis hatékonyak és jól alkalmazhatók a gyakorlatban.

3.2.1. Bounding box based collision

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. Az alábbi ábra ezt mutatja be:



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

Legegyszerűbb esetben a befoglaló dobozt az objektum kétdimenziós képe, textúrája határozza meg. Ez betöltéskor nagyon egyszerűen kiszámítható, hiszen a textúra képének szélessége és magassága lesz abban az esetben ha a Sprite megfelelően van megrajzolva és nem tartalmaz felesleges átlátszó pixeleket a széleken. A doboz meghatározásánál általában törekednek arra, hogy a legjobban illeszkedőt (Best fit rectangle) állapítsák, vagy adják meg. Ennek oka az, hogy csökkentsék/elkerüljék a fals ütközéseket. Gondoljunk csak bele abba, ha a fenti objektum esetében az x tengely mentén megnövelnénk a doboz méretét, akkor azt eredményezné, hogy akkor is érzékelnénk ütközést, amikor még nem értünk a falhoz.

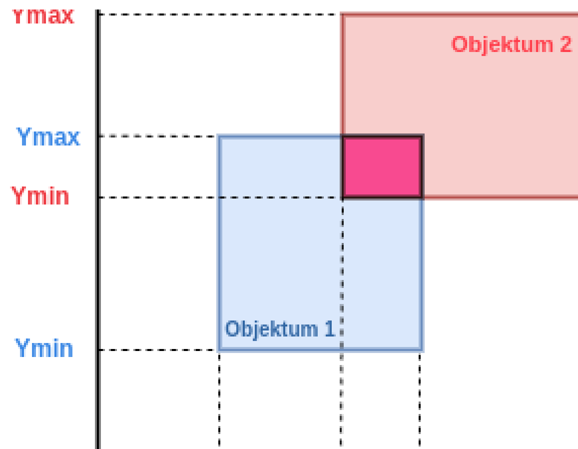
A következő kódrészlet egy lehetséges befoglaló dobozt megvalósító osztály leírást mutat be:

```
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 egy befoglaló dobozt a 4 sarokponttal lehet megadni (*bbPoints[4]*), de a későbbi számítások gyorsítása érdekében célszerű tárolni a képernyő koordináta rendszeréhez viszonyított minimum (*minpoint*), illetve maximum pontját (*maxpoint*). A fenti ábrán ez a bal alsó és a jobb felső pontot jelenti. Mindezeken felül a számításokhoz szükség lesz egy transzformációt elvégző mátrix osztályra, gyorsítási szempontból pedig a doboz szélességének és magasságának a felét is érdemes eltárolni. Mivel a kép bal alsó pontját használjuk a textúra koordinátatér origójaként, ezek az értékek szükségesek a középpont alapú elforgatás során.

Az objektum transzformációja (eltolás, forgatás, nyújtás) során a doboz koordinátáját szintén transzformálni kell. Ezt akkor kell megtennünk, amikor az objektum mozgásakor kiszámítjuk annak új pozícióját. Másik megoldás még az lehet, hogy a BB pontjait akkor számítjuk ki, amikor a program használni fogja azt (pl. ütközésvizsgálat, képernyő vágás, stb), azonban ilyenkor a többszöri kiszámításhoz több erőforrásra van szükség.

Az ütközést meghatározó algoritmust nagyon egyszerű megfogalmazni: amikor két objektum befoglaló doboza átfedi egymást, az objektumok ütköznek. A következő ábra ezt szemlélteti:



Ábra 4. AABB alapú ütközés

Jól látható, hogy a befoglaló dobozok átfedéséből egyértelműen meghatározható az ütközés ténye. Az átfedés kiszámítása alapvetően nem teljesítmény igényes művelet. Azonban ha sok objektum van a képernyőn, akkor már jelentős energiával, CPU idővel kell számolnunk. Ezért megvalósítási szempontból érdemesebb azt az esetet vizsgálni, amikor nincs ütközés. Ez kicsivel gyorsabb megoldást 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;
}
```

Meg kell említeni a befoglaló doboz alapú megoldás hibáját is. Amennyiben olyan objektumok ütköznek egymásnak, amik „lyukasak”, és valójában a lyukas részek fedik csak át egymást, úgy nem történik tényleges ütközés, mint ahogy az a fenti képen is látható. Bár nem közelítik jól az objektumot, mégis hatékonyak és jól alkalmazhatók a gyakorlatban. A hiba ellenére a játékfejlesztés területén ez a megoldás terjedt el leginkább.

Jelen cikk nem terjed ki a befoglaló doboz elforgatására. Míg az OBB (Oriented Bounding Box) típusú megoldás igazán jó eredményeket ad [17], addig a gyakorlatban az AABB alapú forgatás is kielégítő a legtöbb esetben. Az AABB-k esetében a dobozt (sarkait) is el kell forgatni és forgatás közben számolni. Az új négy sarokpontból pedig ismét egy AABB építhető az objektum köré. Hátránya, hogy a doboz mérete változik, így ez befolyásolja az ütközési teszt pontosságát.

3.3. Sprite animáció

Az animáció fontos szerepet tölt be a számítógépes vizualizációban. Ettől válik igazán „élővé” az alkalmazás legyen az egy menü, ablak vagy egy ugráló figura animációja. A számítógépes játékok esetében különösképpen nagy hangsúlyt

fektetnek a folyamatos, minőségi mozgások kidolgozására.

A klasszikus, jól ismert megoldása egy animáció létrehozásának lényegében nem más, mint egy az animációt megvalósító képek halmazának (fázisok) megadott szekvenciában és sebességgel történő váltogatása. A képeket nevezhetjük a memóriába betöltött textúrák egy tömbjének, amely tartalmazza az animáció minden egyes fázisát. Sokan ezt az textúrákból álló objektumot Sprite-nak is nevezik. Minél több fázist tartalmaz a tömb, annál folytonosabb lesz a megjelenítéskor az objektum animációja.

A fájlrendszerben az animáció képei többféle módon tárolhatók. A leginkább elterjedtebb megoldás, amikor egy nagyobb képben tároljuk az egyes frame-eket egymás mellett (*spritesheet*). A következő kép ezt illusztrálja:

Ábra 5. Minta spritesheet

A készítők kiválasztanak valamilyen egységes háttérszínt, és az animációkat egymás mellé helyezve tárolják. Majd betöltéskor ezeket külön textúra objektumokra vágják szét. Az ilyen jellegű kétdimenziós rajzokat összefoglaló néven „**Pixel Art**”-nak (Pixel grafikának) nevezik, mert nagyrészt kézzel készülnek pixelről pixelre rajzolva. Bár a modern háromdimenziós grafika nagymértékben átalakította a számítógépes vizualizációt, mind a mai napig számos pixelgrafikus játék készül.

```
class CSprite {

  vector<CSpriteFrame*> mFrames;        // Frames vector
  CVector2 mPosition;                   // Position of the sprite
  CVector2 mScale;              // Scale of the sprite
  int mNumFrames;               // Number of frames
  int mActualFrame;             // Actual frame
  unsigned int mLastUpdate;     // The last time the animation was update
  unsigned int mFps;            // The number of frames per second
  float mRotationZ;             // Z oriented Orientation of the sprite
  string mName;                 // Name of the Sprite
  bool mLoopAnimation;          // Loop animation

public:

...
};
```

Ahogy a kódból látszik, a *CSprite* osztály nem más, mint a frame-eket és az osztályhoz tartozó tulajdonságokat tároló egység. Minden animációs fázist egy *CSpriteFrame* reprezentál, melyet részletesebben később ismertetünk. A további adatok pedig az olyan fontos tulajdonságokat tárolnak, mint a sprite pozíciója (*mPosition*), mérete (*mScale*), neve (*mName*), frame-ek száma, valamint olyan egyéb adatok, amelyek majd a fázisok tényleges egymás utáni mozgatásához lesznek szükségesek. A név azért fontos, mert sokkal könnyebb egy programban névvel hivatkozni egy animációs fázisra, felhasználó barátta teszi azt. Például „kérem az engine-től a „running” fázist”.

Jól látszik, hogy az animációs fázisok nem közvetlenül egy *CTexture2D* alapú vektorban tárolódnak, hanem egy *CSpriteFrame* osztályon keresztül kerül megvalósításra. Felmerülhet a kérdés, hogy valójában miért van szükség erre? A válasz a befoglaló dobozokhoz kapcsolódik. A játék objektumok fázisai különböző méretűek lehetnek (pl. lehasal a főhős a földre, ugrás, lövés). Magától értendő, hogy az ütközésdetektálást majd ennek megfelelően kell elvégezni. Mivel a *CTexture2D* osztály egy alapegység, így nem célszerű azt felruházni az ütközés érzékelést segítő valamilyen befoglaló objektummal. Ezt a feladatot tölti be a *CSpriteFrame* osztály, amelynek egyszerű struktúrája az alábbi:

```
class CSpriteFrame {

  CTexture2D* mFrame;                 // Frame texture
  string mName;                       // Name of the frame
  CBoundingBox2D* mBBoxOriginal;      // Original Bounding box
  CBoundingBox2D* mBBoxTransformed;   // Transformed Bounding box

public:
  CSpriteFrame();

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

A *CSpriteFrame* osztály tehát a különböző frame-k tárolásáért felelős. A képet egy *CTexture2D* típusú osztály tárolja. Célszerű tárolni továbbá a frame nevét, esetleg szükség lehet rá bizonyos helyzetekben. Az utolsó fontos adat a frame befoglaló dobozai, amelyekre az ütközésvizsgálat során lesz szükség. A doboznak két változatát tároljuk: az egyik az eredeti, a másik pedig a transzformált. Mindkettő tárolásának fontossága a teljesítményben keresendő. A játékmotornak többször is szüksége van az objektum dobozaira/dobozára. Mivel az objektum foroghat, így a doboz is transzformálódik majd vele. Emiatt célszerű tárolni az aktuális transzformált eredményt is.

Bár eddig kizárólag a játékobjektumok animációi voltak előtérben, maga a *Sprite* osztály éppúgy alkalmas a mozgó GUI elemek leírására is. Jó példa erre egy olyan gomb, amely fölé az egér kerül, akkor megváltozik. Egy *Button* osztály lényegében egy két (vagy több) animációs fázissal rendelkező Sprite objektumra épülő elem lehet.

3.3.1. Animációt leíró fájl

A *sprite* betöltési folyamatot egy leíró fájl alapján megtervezni és támogatni. Ez a megközelítés támogatja a korábban már hangsúlyozott rugalmas programozói API tervezési koncepciót. A formátum alapjául célszerű valamilyen olyan ismert tárolási formát választani mint a például a JSON, vagy az XML. Egy hatékonyan használható mintaformátum lehet a következő XML forma:

```
<?xml version="1.0" encoding="utf-8"?>
<animation name="Yoshi_anim">
        <sprite numofframes="4">
                <frame name="Yoshi_Anim_Start" file="yoshi1.tga">
                        <aabb minx="0" miny="0" maxx="64" maxy="64" />
                </frame>
                <frame name="Yoshi_Anim_Start" file="yoshi2.tga">
                        <aabb minx="0" miny="0" maxx="64" maxy="64" />
                </frame>
                <frame name="Yoshi_Anim_Start" file="yoshi3.tga">
                        <aabb minx="0" miny="0" maxx="64" maxy="64" />
                </frame>
                <frame name="Yoshi_Anim_Start" file="yoshi4.tga">
```

```
            <aabb minx="0" miny="0" maxx="64" maxy="64" />
        </frame>
    </sprite>
```

A példa egy XML alapú leíró formátumot mutat be. A formátum láthatóan egyszerű: tetszőleges számú frame-et definiálhatunk, amelyekhez pedig befoglaló dobozt és nevet is társíthatunk. Ne felejtsük el azonban, hogy a leíró önmagában még nem lesz elegendő egy komplexebb objektum animációinak tárolására, hiszen az objektum egy fázisát képes csak tárolni.

3.4. A játék objektum

Önmagában a *Sprite* osztály még nem elég mindenre. Ahhoz, hogy egy játék programozása során a játékelemeket, objektumokat kényelmesen, magas szintű elemekkel tudjuk leírni, ahhoz szükség van egy „játékobjektum” osztályra. A *Sprite* osztályra továbbra is fontos szerepet tölt be, hiszen felhasználhatjuk például GUI elemek (pl. Animált gombok, stb) létrehozására, vagy amire mi is szeretnénk, a tényleges játékra szánt objektumok alapjául. Egy kétdimenziós számítógépes játékban egy játék objektumnak nem csak 1 animációs fázisa van, hanem annyi, ahány állapotot fel tud venni. Például a főhős képes futni, dobni, ugrani, stb. Ha jobban meggondoljuk, akkor egy olyan magasabb szintű osztályra van szükség, amely Sprite-ok tömbjét foglalja magába, és az objektum állapotától függően (séta, guggolás, stb) ezek válthatók. Mindezeken felül természetesen további állapotváltozók és metódusok bevezetésére is szükség van. Egy játék objektum példa implementáció a következő lehet:

```
class CGameObject2D {

CString mName;                  // Entity Name
CVector2 mPosition;             // Position of the object
CVector2 mDirection;            // Direction of the movement
CVector2 mScale;                // Size value
vector<CSprite*> mAnimations;   // Animation
float mSpeed;                   // Speed of the object
float mRotation;                // Rotation value
bool mVisible;                  // Visible or not
bool mCollidable;               // Object is collidable or not
bool mInFrustum;                // Object is in screen frustum or not
unsigned int mCurrentAnim;      // Current Animation Frame
unsigned int mNumberOfFrames;   // Number of Animations
unsigned int mID;               // ID of the Object
int mZindex;                    // z index of the object
C2DGraphicsLayer* mParentLayer; // Parent layer of the object

public:
...
};
```

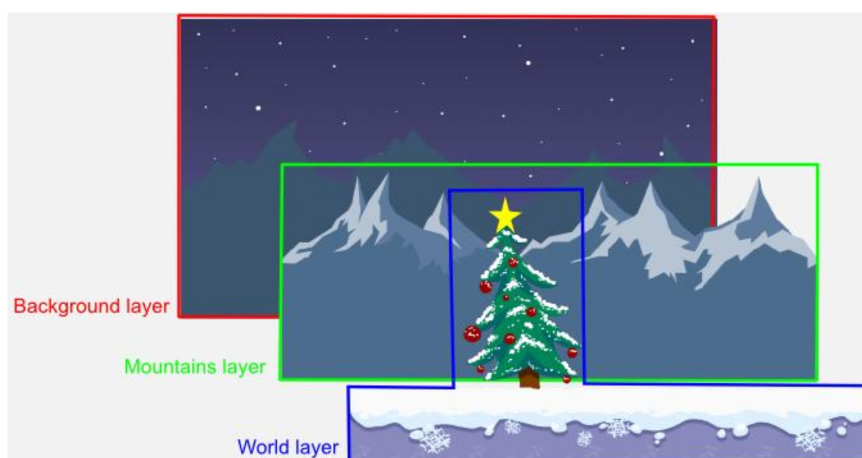
Az első és legfontosabb, amire szüksége van egy *GameObject* osztálynak az a név (*mName*) és az azonosító (*mID*), ugyanis ezekkel tudunk majd hivatkozni rá. Önmagában az *mID* elég lenne, de kényelmi szempontból jobb, ha névvel is tudunk hivatkozni az objektumra. Természetesen egy objektumnak szüksége van a továbbiakban pozícióra, irányra, és a méretét meghatározó tulajdonságokra. Az irányt felhasználhatjuk arra, hogy egyfajta automatikus mozgatást valósítsunk meg. Tehát a felhasználó megadja az irányt és a sebességet (*mSpeed*), és a játékmotor elvégzi az objektum célba juttatását. Az objektumhoz tartozó animációs fázisokat a Sprite-ok tömbje (*mAnimations*) tárolja, az objektum elfordulását pedig az *mRotation*. Bár a vizualizáció optimalizálását a későbbiekben tárgyaljuk, a láthatóság szempontjából itt két hasznos adat is megjelenik. Az *mVisible* segítségével a programozó szabályozhatja, hogy az adott objektum-ot kell-e

rajzolni vagy sem. Az *mInFrustum* adattag pedig egy belső változó arra a célra fenntartva, hogy eltárolja azt az információt, hogy a motor meghatározza, hogy az objektum benne van-e a képernyő tartományába.

Egy különösen hasznos adattag az *mZindex*, amely segítségével az objektumok kirajzolásának sorrendje határozható meg. Bizonyos helyzetekben az objektumok átfedhetik egymást, amelynek általában valamilyen programozói logika által meghatározott sorrendet jelentenek. Vannak olyan objektumok (pl. Felhő), amely rárajzolódik például a hely objektumra. Ennek megvalósítása megkövetel egy numerikus érték bevezetését (pl. *z* érték) amely ezt a sorrendiséget hivatott reprezentálni. 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 azonban egy további kiegészítést igényel, miszerint kirajzolás előtt az objektumokat *z* értéke alapján rendezni kell. A kirajzolás megfelelő sorrendje így biztosítható.

4. Layering rendszer kialakítása

Ha megfigyelünk több népszerű kétdimenziós játékot, akkor felfedezhetjük, hogy a képernyő bizonyos részei, a háttér, esetleg néhány nem játék objektum más sebességgel mozog. Gyakorlatilag kimondhatjuk azt, hogy az elemek úgynevezett logikai rétegekbe vannak szervezve. Nem mondható ki, hogy minden játék így működik, de az tény, hogy a logikai rétegek bevezetése nagymértékben segíti a játékvilágok felépítését, az objektumot mozgatását, stb. A leggyakoribb felosztás egy egyszerűbb játék esetében az, ha elkülönítünk egy réteget a pálya háttérnek (pl. super mario), egy réteget a valós játék objektumoknak (pl. mario, teknősök, stb), és esetleg egy réteget az előtérben mozgatott dolgoknak. Természetesen bármennyi réteg megadható növelve ezzel a játék vizuális megjelenését. Az alábbi képen három logikai réteg alapú megvalósítást láthatunk:



Ábra 6. Három réteget megvalósító játék színtér

Ezek alapján megtervezhető egy olyan osztály, amely képes a rétegek tárolására és kezelésére. Az osztály megvalósítása az alábbi lehet:

```
class C2DGraphicsLayer {

  vector<CGameObject2D*> mObjectList;
  vector<CTexture2D*>      mTextures;
  vector<CParticleSystem2D*> mParticleSystems;  // 2D Particle systems
  bool mVisible;                                // Is layer visible or not
  bool mEnableCollision;                        // Enable Collision on Layer
  CCamera2D *mCamera;                           // Camera for the layer
  CString mName;                                // Layer name
```

```
    int mID;                                    // Layer ID
    C2DScene* mParentScene;                     // Parent Scene

  public:
  …

  };
```

Az értelmezése nagyon egyszerű: szükség van egy olyan listára (*mObjectList*), amely a játék objektumokat tárolja. Mindezek mellett célszerűnek tartjuk egy olyan listának (*mTextures*) a bevezetését is, amely kizárólag 2D textúrákat tárol. Ezek olyan egyszerű képek, mint például a játék háttére, felhők, stb, amelyek nem animáltak. Bár maga a játék objektum is képes lenne ezek tárolására, úgynevezett egy fázissal rendelkező Sprite-ként, maguk az egyszerű textúrák tárolása egyszerűbb és hatékonyabb. Mivel a réteg célja, hogy gyakorlatilag bármilyen olyan elemet tárolhasson, amelye megjelenhet a 2D világban, így olyan elemeket is megjelenítettünk az osztályban, amelyek még nem kerültek tárgyalásra. Ilyen a részecskerendszer (*mParticleSystems*), a tile map (*mLayerTileMap*), és annak a játéktérnek a hivatkozását (*mParentScene*), amelyhez maga a réteg fog tartozni. Ezeken felül a láthatóságot ki-be kapcsoló tulajdonság (*mVisible*), a réteg neve (*mName*), és azonosítója (*mID*) feltétlenül szükséges.

A fenti layer-ing technikával már megvalósítható az úgynevezett "*parallax scrolling"* effekt, amelyet a kétdimenziós játékok előszeretettel alkalmaznak a vizuális élmény növelése érdekében. Alapötlete, hogy a játék világát rétegekre bontják, majd a rétegeket különböző sebességgel mozgatják. Ennek segítségére szolgál a kódban a *CCamera2D* osztály, amely az adott réteg kamerája. Lehetővé teszi, hogy ez réteg "mozgatását" úgy, hogy valójában nem a grafikai objektum kerül mozgatásra, hanem a kamera (nézeti mátrix). Hasonlóan mint egy stratégiai játék esetében, ahol a térkép bejárható (Warcraft, stb) az egér mozgatása segítségével.

A kamera bevezetésével egy újabb probléma jelenik meg. Mivel az objektum (*mPosition*) tényleges koordinátája nem változik meg, csupán rajzoláskor a ézeti (view) matrix által más helyre kerül, így be kell vezetni egy olyan metódust, amely képes kiszámolni az objektum virtuális térbeli helyét olyan CPU feladatok számára, mint például az ütközés detektálás. Például a platformer játékok esetén a főhős mindig a képernyő tartományában marad, míg akár már több képernyőt is arrébb ment a világban.

5. Layering rendszer kialakítása

A rétegek bevezetése sokat segít abban, hogy valóban egy komplexebb játék világot legyünk képesek kezelni. Azonban ezzel még nincs vége, hiszen a rétegeket is célszerű belefoglalni valamilyen struktúrába. Ezek egy lehetséges megvalósítása a játékszíntér (*Scene*). Ha definiálni szeretnénk, akkor a színtér egy az aktuális játékvilágból kiragadt részletet foglal magában. Ez akár lehet egy teljes pálya, de annak akár egy részlete is. Ha még emlékszünk a klasszikus super mario játékra, akkor megfigyelhettük, hogy bizonyos csöveken lecsúszhatunk a föld alá. Legegyszerűbben ezt úgy lehetne megvalósítani, ha az aktuális pályát több színtérből készítjük el: egy a szokványos pályának, valamint minden egyes földalatti részeknek egy külön színtér.

A színtér megvalósítása nagyon egyszerű, mert valójában egy befoglaló logikai struktúra szerepét tölti be. Az alábbi kód egy ilyen mintát mutat be:

```
class C2DScene {

  CVector<C2DGraphicsLayer*> mLayers;      // Layer for objects
  CString mName;                           // Name of the scene
```

```
  bool mVisible;                          // Visibility flag
  CCamera2D *mSceneCamera;         // Global camera for the whole scene

public:
…
}
```

A scene osztály különböző rétegeket (*mLayers*) foglal magába, hogy a játék különböző részei szintén élvezhesség a réteges megvalósítás előnyeit. Tárolnunk kell a scene nevét (*mName*), melynek a korábbiakhoz hasonlóan praktikussági okai vannak. A színtér ki be kapcsolására egy adattagok kell bevezetni (*mVisible*). Ne felejtsük el, hogy attól, hogy supermarió a felső pálya részben ugrált, valójában a föld alatti rész is a memóriában volt, csak nem volt látható. Végül célszerű bevezetni egy scene szintű kamerát (*mSceneCamera*). Bár ugyan minden réteg saját kamerával rendelkezik, de sok esetben merül fel az igény a teljes színtér egyben való mozgatására.

Az osztály nem adattagokban kell, hogy erős legyen, hanem inkább a nyújtott szolgáltatásokban, metódusokban. Tudnunk kell rétegeket hozzáadni, törölni és lekérdezni. Képesnek kell lenni olyan magas szintű szolgáltatásokra, mint a játék objektumok referenciáinak lekérése akármelyik rétegen is vannak, azok elszabadítására ha például törölni kell egy lövedéket, kameramozgások kezelése, *z* alapú rendezés, stb. Ezek nem jelentenek bonyolult implementációt, inkább csak egy kiszolgáló interfészt ad a programozó kezébe. Ez az interfész, maga az osztály használata akkor lesz hatékony, ha kényelmes hozzáférés biztosít a magába foglalt rétegekbe is.

5.1. Színtér kezelő

A színtér megvalósítás akkor válik teljessé, ha azokat menedzselni is tudjuk. Ehhez pedig szükség van egy további befoglaló osztályra. Szerepe gyakorlatilag annyiban merül ki, hogy tárolja (*mScenes*) a színtereket és számos olyan kényelmi funkciót, interfészt biztosít, amellyel a tárolt színtereket, esetleg játék objektumokat érhetjük el akár ID, akár névvel hivatkozva. Egy példa implementáció a következő:

```
class C2DSceneManager {

  vector<C2DScene*> mScenes;
  bool mDrawBoundingBox;          // Draw BB or not
  sColor mBoundingShapeColor;     // Color of the bounding shape

public:
  ...
 C2DScene* LoadSceneXML(CString sceneFilename);
 void RegisterScene(C2DScene* scene);
 void Render();
 CGameObject2D* GetObjectByName(CString name);
 C2DScene* GetScene(u32 sceneID);
 C2DScene* GetSceneByName(CString name);
 void FreeObject(u32 id);
 void FreeObjects();
 void Clear();
 void FreeAScene(CString name);
 void FreeASceneByID(u32 sceneID);
 ...
}
```

Az osztály implementációja nem lesz bonyolult, szerepe inkább a kiszolgáló funkciókban merül ki, amelyekből a legfontosabbakat jelen mintában feltüntettük.

5.2. Virtuális világ tárolása

Már a legegyszerűbb számítógépes játék esetén felmerül az igény arra, hogy a virtuális világot ne beégetve tároljuk a különböző osztályokban, hanem valamilyen megoldás szülessen ennek hatékony kezelésére. Kézenfekvő a különböző "pályákat" a fájlrendszerben tárolni, amelynek két módja a bináris és a szöveges tárolási forma. Kezdők számára mindenféleképpen a szöveges forma javasolt, hiszen amikor még nincs kiforrott játékötlet, vagy kód a játék meghajtására, akkor a szöveges állomány alapú tárolás nagymértékben megkönnyíti a folyamatos módosításokat, próbálgatásokat, mivel egy egyszerű szövegszerkesztővel editálható.

Az alábbiakban egy minta virtuális világ XML formájú leírása szerepel:

```
<?xml version="1.0" encoding="utf-8"?>
<scene name="Platformer_Demo_Scene" layers="4" >
	<layer id="0" name="Sky_layer">
		<texture id="0" x="0" y="0" file="sky.pcx" />
	</layer>
	<layer id="1" name="Mountain_Layer">
		<texture id="1" x="0" y="0" file="mountain.tga" />
	</layer>
	<layer id="2" name="Ground_Layer">
		<texture id="2" x="0" y="0" file="ground.tga" />
	</layer>
	<layer id="3" name="Character_layer">
		<gameobject id="777" name="Liza" collidable="1" zindex="0">
			<sprite file="girl.ani" />
			<position x="450" y="565" />
			<direction x="1" y="0" />
			<scale x="1" y="1" />
			<speed value="0" />
			<rotate value="0" />
		</gameobject>
	</layer>
</scene>
```

A minta egy egyszerű világot ír le, amelyben három réteg van és egy mozgatható karakter. A leírásban szereplő *girl.ani* egy animált karaktert ír le, amelynek formátuma a fent vázolt XML animációs leíró. Természetesen a fenti világ leíró csak a legfontosabb elemeket tartalmazza, igény szerint könnyen bővíthető.

6. A játékmotor szíve

Egy szoftver logikai felépítését mindig szoftver-technológiai szempontok alapján kell megtervezni és felépíteni [4]. A szerkezeti felépítés, az osztályok (feltéve ha OO nyelven készül) kapcsolatai nem közelíthetők meg egzaktan. Mivel számos tervezési minta létezik és szubjektív tényezők is nagymértékben befolyásolhatják a felépítést, így nincs olyan egységes irány vagy szabályrendszer, amit mindenféleképpen követni kell a szoftverünk struktúrájának kialakításakor. Természetesen mindig érdemes körülnézni, hiszen jól bevált minták vannak melyek nyelvenként és az alkalmazott keretrendszerek függvényében eltérhetnek egymástól.

A szoftver alapjainak megfelelő megtervezése komoly ilyen irányultságú ismereteket igényelhet, főleg ha saját játékmotorról van szó. A megfelelő környezet kialakítása sokat segít abban, hogy az ebbe elhelyezendő, erre épülő játékosztályok fejlesztése könnyed legyen. Tapasztalataim szerint a nem megfelelő kialakítás egy tipikus jelzője, amikor sokáig kell keresgélni a programkódban, hogy az adott információ honnan származik, mi hívja meg az adott metódust és merre találhatók a logika bizonyos elemei.

A következő egyszerű C++ modell egy példát mutat be a strukturális alapokra, amely arra elég, hogy segítségével el lehessen indulni egy program készítésében. A játékszoftverek ma tipikusan objektum orientáltan készülnek, így felépítésben is hasonló mintákat követnek. A szoftver mélyén valahol mindig az alábbihoz hasonló mintát alkalmaznak:

```
/// Universal Application Class
class App {

  protected:
  CEngine *mEngine;    // Engine Class as member variable

  public:
  App();
  virtual ~App();
  virtual void Startup() = 0;
  virtual void Run() = 0;
  virtual void Shutdown() = 0;
  virtual void ResizeWindow(int width, int height) = 0;
};
```

A fenti osztály egy általános keretet biztosít a leendő alkalmazásunk számára. Mivel a metódusok láthatóan virtuálisak, így azokat majd a gyermek osztály fogja megvalósítani. A konstruktorban elhelyezett példányosítás pedig a motor automatikus inicializálását teszi lehetővé, a destruktor pedig a felszabadítást:

```
CApp::CApp(){
    mEngine = nullptr;

    // Allocate Memory for the ENGINE
    mEngine = new CEngine();

    if (mEngine == nullptr) {
        printf("\nError: cannot allocate memory for Engine!");
    }
}

CApp::~CApp() {
    if (mEngine != nullptr)
        delete mEngine;
}
```

Nyilvánvaló, hogy mivel a metódusok "pure virtual" minősítővel vannak ellátva, a gyermekosztály fogja megvalósítani őket. Tehát a fenti osztály használatához létre kell hozni egy újabb osztályt (pl. egy játékosztályt) a CApp osztálytól való örökléssel. Az öröklés során ez a fő játékosztályunk közvetlenül megkapja a *mEngine* referenciát, ami hatékony módja annak, hogy a játékmotor által kínált funkciókat hatékonyan tudjuk elérni. Egy példa játékosztály:

MyGame.h

```
class MyGame : public CApp {
public :
        MyGame ( ) ;
        ~MyGame ( ) ;
        void Startup (void);
        void Run (void) ;
        void Shutdown ( void ) ;
        void ResizeWindow (int width , in t  height ) ;
    } ;
```

A példában a "*MyGame*" gyermekosztály az *App* szülő osztály megfelelő metódusait valósítja meg. Az “*Startup*” az alkalmazás közvetlen indítása után szükséges inicializálási részek kezelésére szolgál. Például az ablak megnyitása, egy grafikus kontextus létrehozása és minden más, ami itt van. A "*Run*" metódus a "játékciklus - game loop" megvalósításáért, a folyamatos futásért lesz felelős, a "*Leállítás*" pedig az alkalmazás leállása előtti utómunkálatok elvégzésére van fenntartva. Például memória felszabadítás, grafikus kontextus megszüntetése, erőforrások felszabadítása, stb. Egy további fontos öröklött metódus a "*ResizeWindow*", amelynek feladata az, hogy a programozónak legyen lehetősége az alkalmazásablak átméretezési eseményének kezelésére.

Végül pedig a hiányzó main függvény:

```
#include "App . h"
int main ( int arg c, char * argv [] ) {
    MyGame appliation;
    appliation.Startup( ) ;
    appliation.Run( ) ;
    appliation.Shutdown( );
    return 0;
}
```

Bár az ebben a formában bemutatott minta már alkalmas alkalmazások fejlesztésére, összetettebb szoftverek és grafikus motorok esetén természetesen több irányban szükséges bővíteni. Ezért a következőkben áttekintjük a fenti kód alrendszerekkel való kiterjesztését, például megtudhatjuk, miért fontos a szülőosztály említett metódusaihoz a "virtuális" minősítőt hozzárendelni.

6.1. Alrendszerek és komponensek

Egy számítógépes játék vagy játékmotor számos különböző, egymással kommunikáló alrendszerből és komponensből tevődhet össze. Amikor a motor elindul, mindegyik alrendszert inicializálni kell általában egy előre meghatározott sorrend alapján. Bizonyos komponensek pedig egymástól is függhetnek. Például ha B alrendszer függ az A alrendszertől, akkor elsőként az A alrendszert kell elindítani és konfigurálni. Leállítás esetén pedig gyakran ennek a fordítottja kell megvalósuljon. Egy ilyen komplex szoftver helyes megvalósítása tervezési minták alkalmazását igényli. A gyakorlatban számos minta (*Command*, *Flyweight*, *Observer*, *Prototype*, *Singleton*, *State, stb*) szolgál alapul a struktúra kialakítására. Egy játék vagy motor általában több mintát is alkalmaz attól függően, hogy milyen igényeket kell kiszolgálni. A következőkben egy rövid példán keresztül bemutatjuk, a fenti példa hogyan egészíthető ki egy hatékonyabb, kicsit motor szerű komplexebb struktúrává.

Általános elvárás, hogy a szoftver logikai részeit úgy kell tervezni, hogy azok önálló komponensek formájában jelenjenek meg minimalizálva a függőséget más részektől. Célszerű a tervezés során először a magasabb szintű egységek felől a kisebb egységek irányába haladni. Tehát elsőként az alrendszereket kell tisztázni, majd ezek használatához szükséges olyan átfogó működtető keretet, amibe az

alrendszerek beágyazódnak.

Elsőként vizsgáljuk meg, hogy egy alrendszer milyen elvek mentén épülhet el.

```
class CComponent {

protected:

   int mID;            // Unique id of the component
   string mName;       // Name of the component

   public:

   CComponent();
   virtual ~CComponent();

   int getID(void);
   void setID(int id);
   void SetName(string name);
   string GetName(void);

   virtual void Startup(void) = 0;
   virtual void Update(void) = 0;
   virtual void Shutdown(void) = 0;
   virtual void HandleMessage(void) = 0;
};
```

Az alrendszerek felépítésében vannak közös pontok, ezért ezeket célszerű kiemelni egy külön osztályba. Jelen példában a *CComponent* biztosítja nekünk azokat a (jelenleg minimális) fő funkciókat, amiket első megközelítésünk alapján minden alrendszernek tudnia kell. Minden olyan osztály, amely ebből fog származni, rendelkezni fog egy azonosítóval (*mID*) és egy névvel (*mName*), valamint ezek lekérdező és beállító metódusaival. Az adattagok mellett szükség van inicializáló(*Startup*), adatokat frissítő (*Update*), leállító (*Shutdown*) és üzeneteket kezelő (*HandleMessage*) metódusokra is, melyek láthatóan virtuálisak és törzsrész nélküliek, így ezeket majd a gyermek osztályban kell megvalósítani. Ezzel lehetővé fog válni, hogy az összes kompenst egységesen lehessen kezelni (elindítani, leállítani) a későbbiekben.

Az ős megfelelő kialakítása után az alrendszerek kialakítása következik. Az alábbi minta egy példát mutat be alrendszerek definiálására:

```
#include "CComponent.h"

class CSampleSubsystem : public CComponent {

public:

    CSampleSubsystem();
    ~CSampleSubsystem();

    // Component inherited tasks
    void Startup(void);
    void Update(void);
    void Shutdown(void);
    void HandleMessage(void);
};
```

Az alrendszer felépítése a jelenlegi szinten hasonlóan egyszerű. Az implementáció során kötelezően meg kell valósítani az örökölt virtuális metódusokat, melyekre majd a későbbi, az alrendszereket egységbe fogó vezérlő osztályban lesz szükség, valamint azokat a metódusokat, amelyek már alrendszer specifikusak lesznek (Pl. grafikus kontextus inicializálása, hangrendszer konfigurációja, stb). A jelenlegi mintában nincs specifikus rész.

A továbbiakban szükség van legalább egy olyan központi elemre, osztályra, amely

az egész logikai szoftver struktúra központi elemeként fog szolgálni. Összefogja az alrendszereket és olyan további fontos funkciókat biztosít, mint az alrendszerek inicialzálása, leállítása, és az úgynevezett "main loop" (*game loop*) biztosítása. A mi megközelítésünk szerint ez a *CEngine* osztály lesz.

```
class CEngine
{
    // List of subsystems
    vector<CComponent*> mComponents;

public:

    static GraphicsManager*   gGraphicsManager;  // Graphics Manager Subsystem
    static CInputManager*     gInputManager;     // Input Manager subsystem
    static CShaderManager*    gShaderManager;    // Shader Manager subsystem
    static CTexture2DManager* gTextureManager;   // Texture Manager
    static C2DSceneManager*   g2DSceneManager;   // 2D Scene Manager
    […]

    CEngine();
    ~CEngine();

    void Shutdown();                             // Shutdown
    void MainLoop();                             // Main loop of the engine
    void RegisterSubSystem(CComponent *system);  // Register a system to Engine

private:

    bool InitSubSystems();        // Init all built in subsystem
};
```

A *CEngine* osztály nagyon fontos része a programnak. Feladata, hogy összefogja és vezérelje a rendelkezésre álló komponenseket. Itt szerepel az úgynevezett *MainLoop / Game Loop*, amely egy feltételek mellett értelmezett végtelen ciklus. Itt kerülnek elvégzésre az olyan alapvető funkciók, mint az input kezelés, az állapotok frissítése, eltelt idő mérése, az objektumok kirajzolása és mozgatása, képernyő frissítése, stb. A gyakorlatban, egy valós alkalmazásban való közvetlen hivatkozás miatt célszerű az alrendszereket külön, statikus egységként is kezelni. Statikus mivoltuk révén hivatkozásuk a kód bármely részén így nagyon egyszerűvé válik. Például a grafikus menedzser elérése: *CEngine::gGraphicsManager*. Ez a megközelítés gyakorlatilag az ismert *„Singleton"* tervezési mintának felel meg, miszerint csakis egy példány létezhet az adott osztályból.

7. Összefoglalás

A számítógépes játékfejlesztés egy komplex ismereteket igénylő folyamat, amelyben szükség van szoftvertervezési, algoritmizálási és grafikai szaktudásra is. Ezt a tudást a játékmotorok próbálják összefogni és rendszerezni, így pedig hatékony eszközt adni a fejlesztők kezébe. Azonban aki szeretné megérteni a valódi működést a háttérben, nincs más út, mint hogy saját maga építi fel azokat az alapvető komponenseket, amelyekkel már bizonyos játékok elkészíthetők. Ezen az úton olyan mértékű programozói és egyéb tapasztalat szerezhető a komplexebb szoftver rendszerek építésében, amely más úton nem. Célszerű lenne minden játékfejlesztőnek ezen út bejárásával kezdeni, majd csak a későbbiekben fordulni esetleg a játékmotorok világához.

Hivatkozások

[1] Akenine-möller, T., Haines, E.: Real-Time Rendering. A. K. Peters. 3nd Edition, 2008.

[2] MARÍN-LORA, C.; CHOVER, M.; REBOLLO, C., REMOLAR, I.: A game development environment to make 2D games, Communication Papers, Vol.9 – No18, pp. 7-23, 2020.

[3] Jason Gregory: Game Engine Architecture, A K Peters/CRC Press; 3rd edi- tion, 2018.

[4] Eric L.: Foundations of Game Engine Development, Terathon Software LLC, 2019.

[5] Nicolas A. B.: Hands-On Unity 2022 Game Development: Learn to use the latest Unity 2022 features to create your first video game in the simplest way possible,Packt Publishing; 3rd edition., 2022.

[6] Eberly, H. D.: 3D game engine design: A practical approach to real-time computer graphics. CRC Press; 2nd edition, 2006

[7] Anis Zarrad: Game Engine Solutions, Simulation and Gaming, InTech, pp 75-85., 2018

[8] Game Maker, https://gamemaker.io/, 2024.

[9] Unity Engine, https://unity.com, 2024.

[10] Roblox Game Platform, https://www.roblox.com, 2024.

[11] Constructs 3, https://www.construct.net, 2024.

[12] Phaser HTML5 game engine, https://phaser.io, 2024.

[13] Cocos game engine, https://www.cocos.com, 2024.

[14] Godot engine, https://godotengine.org, 2024.

[15] Unreal Game Engine, https://www.unrealengine.com/, 2024

[16] P. Mileff, J. Dudra: The Past and the Future of Computer Visualization, Production Systems and Information Engineering, Volume 10, No 1, pp. 16-29., 2022.

[17] Rong Zhang, Xiaogang Liu, Jianjun Wei: Collision detection Based on OBB Simplified modeling, Journal of Physics Conference Series 1213(4):042079 1213(4):042079, https://doi.org/10.1088/1742-6596/1213/4/042079 , 2019