Production Systems and Information Engineering
Volume 13 (3), pp. 21–32.
https://doi.org/10.32968/psaie.2025.3.3

JÁTÉKCIKLUS: A JÁTÉKMOTOR SZÍVE

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

Absztrakt. A játékipar világa hosszú utat tett meg a Pong és a Space Invaders korszakától napjainkig. Ma a játékok nem csupán szórakoztatási formát jelentenek, hanem egy hatalmas iparágat is, amely számos területen hajtja előre a technológiai fejlődést. Maga a játék egy összetett rendszer, amely számos komponensből áll. Jelen publikációban ezek közül az egyik legfontosabb elemét vizsgáljuk meg: a játékciklus (game loop) fogalmát és annak szerepét a játéktechnológiában. A játékciklus a játékfejlesztés alapvető szerkezeti eleme, amely a játék állapotának folyamatos frissítésével és a grafika kirajzolásával irányítja a játék folyamatát. Célunk, hogy átfogó képet adjunk a különböző típusú játékciklusokról, azok jelentőségéről és működésükről.

Kulcsszavak: *játékfejlesztés, játékciklus, optimalizáció*

1. Bevezetés

A számítógépes játékok napjainkra a szórakoztatás egyik legfontosabb formájává és a mai kultúra meghatározó részévé váltak. Számos előnyt kínálnak, és fontos médiumként szolgálnak szinte minden területen, legyen az egyszerű szórakozás, művészet vagy oktatás. A játéktechnológia egy gyorsan fejlődő terület, amely magában foglalja azokat az eszközöket és technikákat, amelyek segítségével a videójátékokat létrehozzák és futtatják. Ez kiterjed a programozási nyelvekre és fejlesztői eszközökre, valamint a hardver- és szoftverplatformokra, amelyek a játékok futtatását lehetővé teszik. A videójátékok növekvő népszerűségével és a játékipar folyamatos bővülésével a játéktechnológia a kutatás és fejlesztés egyik kulcsfontosságú területévé vált.

A játéktechnológia legújabb fejlesztései számos új lehetőséget nyitottak meg a játékfejlesztők számára: a fejlettebb grafika és a valósághűbb fizika mellett újfajta játékos-interakciók és magával ragadóbb virtuális világok jöttek létre. A gépi tanulás, a mesterséges intelligencia, valamint a virtuális és kiterjesztett valóság használata új kapukat nyitott a játékosok számára nyújtott élmények terén. A többjátékos módok is egyre népszerűbbé váltak, lehetővé téve, hogy a világ minden tájáról érkező játékosok valós időben kapcsolódjanak és versenyezzenek egymással.

Ahogy a játékipar tovább növekszik és fejlődik, a játéktechnológia egyre fontosabb szerepet játszik a médium jövőjének formálásában. Legyen szó új hardver- és szoftverplatformok fejlesztéséről, vagy a játékos-interakció és játékmenet új formáinak megalkotásáról, a játéktechnológia továbbra is a játékipar innovációinak egyik fő motorja marad.

Jelen publikáció célja, hogy átfogó képet adjon a játékok egy rendkívül fontos összetevőjéről. Bemutatjuk a játékciklus jelentőségét, típusait, működésüket, valamint előnyeit és hátrányait.

2. A játékciklus

A játékmotor egy összetett rendszer, amely a videójátékok fejlesztésének és futtatásának alapját képezi [1]. Ez szoftverkomponensek, eszközök és különböző könyvtárak gyűjteménye, amely biztosítja a fejlesztők számára a szükséges funkciókat a játékok elkészítéséhez, teszteléséhez és telepítéséhez. A motor számos technikai feladatot kezel a játékfejlesztés során, például a grafika renderelését, a memória kezelését, valamint a be- és kimeneti műveleteket [2].

A játékmotor egyik kulcseleme a game loop (játékciklus), amely valós időben irányítja a játék folyamatát. Ez felelős a játékállapot frissítéséért, a játékos bemeneteinek kezeléséért és a játék képernyőre való kirajzolásáért [3]. A szoftver ezen része általában egy végtelen ciklusként van implementálva, amely folyamatosan fut a játék ideje alatt. Ismétlődően feldolgozza a bemeneteket, frissíti a játék állapotát, kezeli a fizikát, a mesterséges intelligenciát és megjeleníti a grafikát, ezzel biztosítva a folyamatos animáció és interakció illúzióját. A játékciklus alapvető koncepció a játékfejlesztésben, és szinte minden videójátékban jelen van [4]. Többféle típusát is alkalmazzák, mindegyiknek megvannak a maga előnyei és hátrányai, a választás pedig a játék és a célplatform specifikus követelményeitől függ [7] [8].

A legismertebb game loop típusok:

1. **Fix időlépéses játékciklus (Fixed-time game loop):** Ebben a típusban a ciklus rögzített időközönként fut, függetlenül attól, mennyi idő telt el az utolsó frissítés óta. Ez biztosítja, hogy a játék minden eszközön azonos sebességgel fusson, és csökkenti a változó képkockasebességből adódó hibákat. Hátránya, hogy problémát okozhat, ha a játék túl komplexszé válik, vagy a játékos eszköze nem elég erős a fix framerate kezeléséhez [9].
2. **Változó időlépéses játékciklus (Variable-time game loop):** Itt a játékállapot frissítése az utolsó frissítés óta eltelt idő alapján történik. Ez segíthet a játékot szélesebb eszközkínálaton zökkenőmentesen futtatni, viszont problémát okozhat, ha a framerate túl alacsonyra beesik.
3. **Félig fix időlépéses játékciklus (Semi-fixed time game loop):** Ez a megközelítés a fix és változó időlépéses modellek elemeit ötvözi. A játéklogika fix időlépést használ, míg a renderelést az eltelt időhöz igazítja.
4. **Eseményalapú játékciklus (Event-based game loop):** Ez a típus nem fut folyamatosan, hanem megvárja bizonyos események (pl.: felhasználói input, szerver-

frissítés) bekövetkeztét, majd azokat dolgozza fel. Különösen hasznos lehet olyan játékoknál, ahol alacsony frissítési rátára van szükség.

5. **Aszinkron játékciklus (Asynchronous game loop):** Ez a típus lehetővé teszi, hogy a játék aszinkron módon frissítsen és rendereljen, tehát a játék folytathatja a frissítést és a megjelenítést akkor is, amikor éppen bemenetekre vagy más erőforrásokra vár. Ez csökkentheti a késleltetés érzékelését és reszponzívabbá teheti a játékot.

A ciklus minden egyes iterációját frame-nek nevezzük. A valós idejű játékok másodpercenként többször frissítenek: a két leggyakoribb frissítési ráta a 30 FPS és 60 FPS. Ha egy játék 60 FPS-sel fut, az azt jelenti, hogy a játékciklus másodpercenként 60-szor ismétlődik meg.

2.1. A játékciklus fázisai

A játékciklus (vagy fő ciklus) általában egy végtelen ciklus, amely hagyományos értelemben az alábbi részekre bontható [6]:

- **Bemenet kezelése:** A játékciklus a játékos által adott bemenetek (pl.: billentyűzet, egér vagy gamepad) feldolgozásával kezdődik. Ez a lépés felelős a játékállapot frissítéséért a játékos cselekedetei alapján.
- **Játékállapot frissítése:** A játékállapot a korábbi lépésben érkezett bemenetek és más tényezők (pl.: fizika, mesterséges intelligencia) alapján kerül frissítésre. Ez a fázis felel a karakterek pozíciójának frissítéséért, az ütközések ellenőrzéséért és a játék belső logikájának naprakészen tartásáért.
- **Grafika renderelése:** A játékciklus ezután a frissített játékállapotot jeleníti meg a képernyőn. Ez a lépés rajzolja ki a karaktereket, objektumokat és környezetet.
- **Várakozás / alvás:** Miután a grafika kirajzolásra került, a játékciklus meghatározott ideig vár (a frame rate alapján) a következő iteráció megkezdése előtt. Ez biztosítja, hogy a játék különböző eszközökön is következetes sebességgel fusson.

A játékciklus ezeket a lépéseket ismétli újra és újra, egészen addig, amíg a játék be nem záródik. Érdemes megjegyezni, hogy a lépések sorrendje a játéktól és a használt játék-motortól függően változhat. Például egyes játékmotorok a fizikát és az ütközésdetektálást külön szálon kezelik – ezt nevezzük multithreadingnek (többszálúsításnak).

2.2. A játékciklus problémája

Programozási szempontból az alap játékciklus a következőképpen írható le:

```
while game is running
    process inputs
    update game world
    Render world
Loop
```

Ha a ciklus belsejét megfelelő tartalommal töltjük ki, a szoftverünk működőképes lesz. Még mielőtt azt hinnénk, hogy a játékciklus ennyire egyszerű, sajnos ez a megközelítés egy nagyon fontos hibát hordoz. A hardver és az operációs rendszer természetéből fakadóan a ciklus különböző sebességgel fog futni különböző teljesítményű számítógépeken [10]. Gyors gépen gyorsabban, lassú gépen pedig lassabban hajtódik végre. Ennek következtében a játék sebessége nem lesz azonos.

Vegyünk egy példát a mozgásra:

```
newpos(x,y) = currentpos(x,y) + velocity(v)*direction(x,y)
```

A mozgás során minden objektumra minden képkockában elvégezzük a fenti műveletet a játékciklusban, így a mozgás folyamatos lesz. Azonban lassabb gépen a mozgás is lassú lesz, míg gyorsabb számítógépen túl gyors. Ez a jelenség nagyon korai játékoknál (különösen a DOS korszakban) gyakran előfordult. Emiatt szükség van olyan jobb megközelítésekre, amelyek képesek ezt a problémát megoldani.

3. Játékciklus típusai

A mai szoftverek ezért a fentebb vázolt egyszerű megoldás egy módosított változatát használják, amelyet időalapú mozgásnak (Time Based Movement) nevezünk, vagy ennek valamilyen módosított verzióját. Az algoritmus – ahogy a neve is sugallja – az idő dimenziójából indul ki. Az alapötlet a következő: ha képesek vagyunk mérni az eltelt időt két egymást követő játékciklus futtatása között egy nagyobb felbontású (legalább milliszekundumos) órával, akkor kapunk egy faktort, amely segítségével a sebességet szabványosíthatjuk a különböző gépek között. Ez biztosítja, hogy az objektumok ugyanazzal a sebességgel mozogjanak, még eltérő teljesítményű számítógépeken is.

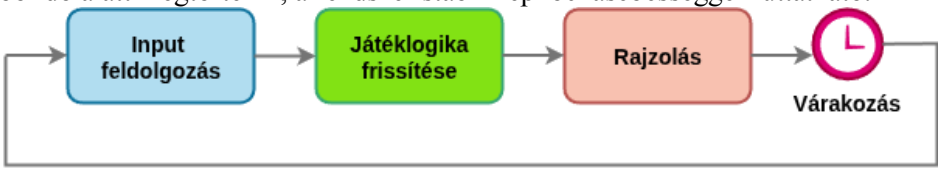
Ez az érték kulcsfontosságú a játékciklus számára, és sok számításban és döntésben szerepet játszik. Gyakori gyakorlat, hogy az eltelt időt használjuk arra, hogy a játék frissítései és animációi függetlenek legyenek a képkockaszámtól és az eszköz teljesítményétől, ezzel biztosítva a konzisztens játékélményt különböző eszközökön és platformokon. Például, ha a játék fizikája fix időlépést használ, az eltelt idő értékéből számítható, hogy az objektumok milyen messzire mozogjanak, és hogyan reagáljanak erőkre vagy ütközésekre.

Az eltelt idő ideális esetben egy dupla pontosságú lebegőpontos szám 0.0 és 1.0 között. Mivel két egymást követő képkocka között biztosan eltelik némi idő, negatív érték nem fordulhat elő. Ha az eltelt idő értéke nulla, akkor a használt időzítő felbontása nem elég kicsi. Ez azt jelenti, hogy a játék implementációjához pontosabb, operációs rendszer szintű órát kell használni, amely kisebb változásokat is képes mérni.

Ezen felül az eltelt idő felhasználható időalapú effektek megvalósításához is, például a játékidő lelassításához vagy felgyorsításához.

3.1. Rögzített időlépéses ciklus

A rögzített időlépéses ciklusok olyan játékciklustípusok, amelyek állandó és kiszámítható frissítési frekvenciát biztosítanak a játékmenet logikájának. Ebben a megközelítésben a játékciklus fix időközönként, például másodpercenként 60 alkalommal fut le. Ez garantálja, hogy a játékmenet logikája és a fizikai számítások meghatározott, állandó ütemben történjenek, ezáltal stabil és előrejelezhető játékélményt nyújtva. Például, ha a cél az, hogy a játék képkockasebessége (FPS) 60 legyen, akkor ez azt jelenti, hogy nagyjából 16 milliszekundum áll rendelkezésre egy játékciklus-iteráció végrehajtására (1000 ms / 60 = 16,6667 ms). Az algoritmus működése a következő: ha a játékciklus az adott időintervallumon belül, vagy annál rövidebb idő alatt végrehajtja a szükséges műveleteket, akkor a ciklus végén várakozik a maradék időtartamig, hogy összesen pontosan 16,6667 ms teljen el. Amennyiben a teljes játékmenet feldolgozása és renderelése megbízhatóan kevesebb idő alatt megtörténik, a rendszer stabil képkockasebességgel futtatható.



1. ábra. Rögzített időlépéses ciklus

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.

A következő kód egy mintapéldát mutat be a játékciklusra, amely ezt az algoritmust használja:

```
double MS_PER_UPDATE = 1.0 / 60.0;
while (true)
{
  double start = getCurrentTime();
  processInput();
  update();
  render();

  sleep(start + MS_PER_FRAME - getCurrentTime());
}
```

A rögzített időlépéses ciklus használatának egyik előnye, hogy segít kiküszöbölni a játék teljesítményének ingadozásait és következetlenségeit. Ez különösen fontos olyan játékok esetén, amelyek pontos időzítést és interakciókat igényelnek, például fizikaalapú vagy gyors tempójú akciójátékoknál. A rögzített időlépéses ciklus biztosítja, hogy az összes számítás következetesen történjen, még akkor is, ha a képkockasebesség változik, ezáltal sima és stabil játékélményt nyújtva.

Egy másik előnye a rögzített időlépéses ciklusoknak, hogy megkönnyítik a játékmenet logikájának fejlesztését és tesztelését. A fejlesztők támaszkodhatnak egy kiszámítható frissítési sebességre, ami egyszerűbbé teszi a hibakeresést és a játékmenet finomhangolását. Ez különösen hasznos lehet olyan játékok fejlesztésekor, amelyek pontos időzítést vagy interakciókat igényelnek.

Fontos megjegyezni, hogy a rögzített időlépéses ciklusoknak lehetnek hátrányai is. Ha az időköz túl rövid, a játék túl megterhelő lehet régebbi vagy gyengébb eszközök számára, ami alacsonyabb képkockasebességhez és kevésbé élvezetes élményhez vezethet. Ugyanakkor, ha az időköz túl hosszú, a játék kevésbé reagál a játékos bemeneteire, ami szintén rontja a játékélményt.

3.2. Változó időlépéses játékciklus

A változó időlépéses játékciklus egy tervezési minta, amelyet videójáték-fejlesztésben használnak a játék frissítési és megjelenítési ciklusainak vezérlésére. Ez a megközelítés nem alkalmaz várakozást a ciklus utolsó lépéseként. Ehelyett az eltelt idő fogalmát használja mérőszámként a teljes játékmenet logikájának frissítéséhez. Ez a módszer segít megelőzni a „lassított” vagy „gyorsított” hatásokat, amelyek akkor jelentkezhetnek, ha a játék lassabb vagy gyorsabb eszközökön fut. Minél hosszabb ideig tart egy képkocka feldolgozása, annál nagyobb lépéseket tesz a játék. A ciklus logikai lépései a következők szerint írhatók le:

```
double lastTime = getCurrentTime();
while (game_is_running)
{
  double current = getCurrentTime();
  double elapsed_time = current - lastTime;
  processInput();
  update(elapsed_time);
  render();
  lastTime = current;
}
```

Minden képkockában meghatározzuk, hogy mennyi valós idő telt el az utolsó játékmenet-frissítés óta (*elapsed_time*). Amikor frissítjük a játék állapotát, ezt az időtényezőt használjuk fel a játék minden mozgásának kiszámításához.

Vegyük ugyanazt a mozgásegyenletet, de most kibővítve az új elapsed_time tényezővel:

```
newpos(x,y) = currentpos(x,y) + elapsed_time * velocity(v) *
                        direction(x,y)
```

Az *elapsed_time* szorzótényezőként működik, ezért az értéke nem lehet nulla. Gyorsabb gépeken ez az idő kisebb lesz, mert a ciklus gyorsabban fut a nagyobb számítási kapacitás miatt, míg lassabb gépeken nagyobb számot kapunk. Ez a tényező tehát kompenzálni tudja a gépek közötti sebességkülönbséget. Vegyük a következő példát:

A játékban kilőnek egy lövedéket, amely átrepül a képernyőn. Lassabb gépeken a newpos(x,y) értéke nagyobb lesz a nagyobb elapsed_time miatt. Gyakorlatilag ez azt jelenti, hogy a lövedék több pixelt „ugrik" egyszerre. Ez persze nem probléma, mert a játék dinamikája ezt elfedi, és csak akkor zavaró a szemnek, ha a lövedék helyzete két fázis között túl nagy távolságra van. Ezzel szemben gyors gépen, az *elapsed_time* kis értéke miatt a lövedék kisebb lépésekben mozog, akár pixelnél is kisebb mozgással (ezért szokás a koordinátákat lebegőpontos *(float)* típusban tárolni). Ilyenkor a mozgás és a játékmenet nagyon sima lesz. Összességében a lövedék (majdnem) ugyanakkora távolságot tesz meg mindkét esetben, csak nem ugyanabban a léptékben.

Ennek a játékciklustípusnak az egyik hátránya, hogy a játék nem determinisztikus lehet. Ez azt jelenti, hogy egy új PC-n a fizikai motor 50-szer frissíti a lövedék helyzetét, míg egy régi PC-n csak tízszer. A legtöbb játék lebegőpontos számokat használ, amelyek kerekítési hibának vannak kitéve. Minden egyes lebegőpontos összeadásnál az eredmény egy kicsit eltérhet a pontos értéktől. A gyors gép ötször annyi műveletet végez, így nagyobb hiba halmozódik fel, mint a régi gépen. Ennek következménye, hogy ugyanaz a lövedék különböző helyeken köt ki a két gépen. Az ütközések pontos kiszámítása így jelentősen nehezebbé válik.

Mindezek mellett a változó időlépéses játékciklus jelentősen javíthatja a játék általános válaszkészségét és az irányítás élményét, különösen azokban a játékokban, amelyek precíz és gyors reagálást igényelnek, mint például platformjátékok vagy első személyű lövöldözős játékok.

A változó időlépéses megoldás közel ideális a mai grafikai alkalmazásokhoz, de ajánlott két kisebb módosítást hozzáadni. A következőkben megvizsgálunk két olyan módosítást, amelyek tovább javíthatják ezt a megoldást.

3.2.1. Eltelt idő korrigálás

Eddig a fent bemutatott megoldás csak az ideális esettel foglalkozott, amikor semmi nem zavarja meg a játékmenetet. A gyakorlatban azonban számos zavaró tényező előfordulhat, például a számítógép akadozása. A játékmenet során előfordulhat, hogy az operációs rendszer bizonyos háttérfolyamatai több erőforrást használnak (például háttérben tömörítünk, vagy az antivírus valamilyen háttérellenőrzést végez stb.), emiatt az elapsed_time értéke többszörösére nőhet, ami azt jelenti, hogy az objektumok sokkal nagyobb mozgásokat végeznek, több pixelt ugranak egy ciklus alatt. Egy tipikus példa erre a hibakeresés (debugging). Amikor a szoftvert megállítjuk hibakeresési célból, majd újraindítjuk, az elapsed_time értéke nagyon magas lesz, mivel a korábbi képkockában mért idő nagyon távol esik a jelenlegitől. Ez a viselkedés, az elapsed_time-nál jelentkező tűhegyszerű kiugrások simátlan játékmenetet eredményeznek.

Ezért érdemes egy felső korlátot beállítani az értékre, például 1.0-ra:

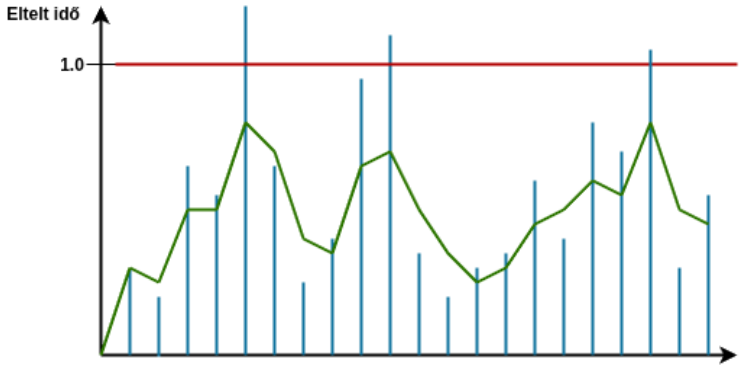
```
if (elapsed_time > 1.0f) {
  elapsed_time = 1.0f;
```

```
}
```

Ezzel a megoldással bár maximalizálhatóak az elapsed_time csúcsértékei, a számítógép terhelése miatti hirtelen, nagy tartományú változás még mindig problémát jelent. Ezek a változások a játékmenetben is észrevehetők. Egy FPS-játék esetén például maga a mozgás is „remeghet”. A változó időlépéses ciklus egyik nagyon hasznos optimalizálása az elapsed_time „simítása”. Egy nagyon egyszerű, de jól működő megoldás, ha az aktuális és az előző *elapsed_time* érték átlagából számoljuk ki a játékmenet frissítéséhez használt értéket:

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

Bár a véletlenszerű terhelések a számítógépen nem szüntethetők meg, hatásuk ezen a módszeren keresztül korrigálható. Ennek eredményeként sokkal simább játékmenet érhető el.



2. ábra. Eltelt idő „simítása”

A 2. ábra egy mintapéldát mutat az elapsed_time értékek lehetséges sorozatára. A piros vonal a felső korlátot jelzi, a zöld érték pedig az aktuális és az előző idő átlagát.

3.3. Félig fix időlépéses játékciklus

A félig rögzített időlépéses játékciklus egy széles körben használt módszer a fizikai szimulációk és egyéb valós idejű folyamatok vezérlésére videójátékokban. Ez a megközelítés ötvözi a rögzített és változó időlépéses módszerek előnyeit, így egyensúlyt teremt a stabilitás és a hatékonyság között. Az algoritmus a szimulációk többségénél rögzített időlépést használ, ugyanakkor bizonyos esetekben, például ha a játék lassabban fut, és fel kell zárkóznia, engedélyezi a változó időlépésű frissítéseket. Ezáltal simább és következetesebb szimulációk érhetők el, miközben hatékonyabbá válik a számítási erőforrások kihasználása.

A félig rögzített időlépéses játékciklus működése a következő:

- **Inicializáció:** A játékállapot és változók inicializálása, beleértve az időlépést (dt) és az akkumulátort (acc).
- **Bemenet kezelése:** A felhasználói input feldolgozása, a játékállapot ennek megfelelő frissítése.
- **Rögzített időlépéses szimuláció:** A játékállapotot rögzített időlépéssel (dt) frissítjük, a szimulációs lépések számát az akkumulátor (acc) határozza meg.
- **Változó időlépéses szimuláció:** Ha az akkumulátor (acc) nagyobb, mint a rögzített időlépés (dt), további szimulációs lépéseket hajtunk végre változó időlépéssel, amíg az akkumulátor értéke kisebb nem lesz, mint dt.
- **Renderelés:** A játékállapot megjelenítése a képernyőn, a frissített szimulációs adatok alapján.
- **Ismétlés:** A játékciklus ismétlése, amíg a játék le nem áll.

4. ábra. Félig fix időlépéses játékciklus

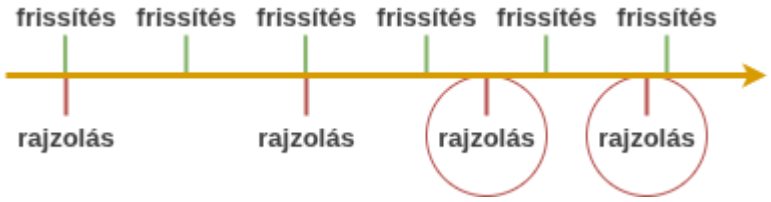
A ciklus a következőképpen írható le:

```
double previous = getCurrentTime();
double frameTime = 0.0;
double MS_PER_UPDATE = 1.0 / 60.0;
while (true)
{
  double current = getCurrentTime();
  double elapsed = current - previous;
  previous = current;
  frameTime += elapsed;

  processInput();

  while (frameTime >= MS_PER_UPDATE)
  {
    update();
    frameTime -= MS_PER_UPDATE;
  }

  render();
}
```

Minden képkocka elején frissítjük a *frameTime* értékét annak alapján, hogy mennyi valós idő telt el. Ez méri, mennyire van lemaradva a játék órája a valós időhöz képest. Ezután feldolgozzuk a bemeneteket, majd elérkezünk a belső ciklushoz, amelynek feladata a játékállapot frissítése rögzített lépésekben, amíg a *frameTime* nagyobb, mint az MS_PER_UPDATE. Végül renderelünk, és a folyamat elölről kezdődik. Minél kisebb az MS_PER_UPDATE értéke, annál több feldolgozási idő szükséges a valós időhöz való felzárkózáshoz. Minél nagyobb, annál darabosabb lesz a játékmenet.

3.3.1. A maradék késleltetés problémája

A maradék késleltetés azt jelenti, hogy a játékot rögzített időlépésben frissítjük, de a renderelés tetszőleges időpillanatokban történik. Ez a felhasználó szemszögéből azt eredményezi, hogy a játék képe gyakran két frissítés közötti állapotot jelenít meg.



4. ábra. A maradék késleltetés problémája

Míg a frissítési fázis rögzített időközönként történik, a renderelési fázis nem determinisztikus. Kevésbé gyakori, mint a frissítés, és nem is állandó. A probléma abból adódik, hogy nem mindig pont a frissítési pontban renderelünk. A 4. ábrán a piros körök jelölik ezeket az eseteket.

Vegyünk egy példát: képzeljünk el egy objektumot, amely áthalad a képernyőn. Az első frissítéskor a bal oldalon van, a második frissítés jobbra mozgatja. A játék azonban egy olyan időpillanatban renderel, ami a két frissítés között van, így a felhasználó azt várná, hogy az objektum a képernyő közepén jelenjen meg. A jelenlegi megoldással viszont még mindig a bal oldalon látszódik. Ez azt eredményezi, hogy a mozgás darabosnak vagy akadozónak tűnik.

Szerencsére pontosan tudjuk, hogy a két frissítési fázis között mennyire vagyunk éppen, amikor a renderelés történik: ezt a *frameTime* tárolja. Az update ciklust akkor állítjuk meg, amikor a frameTime kisebb, mint az update időlépése, nem pedig akkor, amikor nulla. Marad tehát egy „maradék idő”. Ez mutatja meg, hogy mennyire vagyunk a következő *frame*-be lépve. Rendereléskor ezt az értéket adjuk át:

```
render(frameTime / MS_PER_UPDATE);
```

Itt az MS_PER_UPDATE-tel osztunk, hogy normalizáljuk az értéket. A render() függvénynek átadott érték 0 (az előző frame pozíciója) és 1.0 alatti (a következő frame pozíciója) közötti tartományban mozog, függetlenül az update időlépés hosszától. Így a renderernek nem kell a képkockasebességgel foglalkoznia, csak 0 és 1 közötti értékekkel.

A renderer minden játékelem pozícióját és aktuális sebességét ismeri. Tegyük fel, hogy egy objektum 20 pixelre van a képernyő bal oldalától, és jobbra mozog 400 pixelt képkockánként. Ha éppen félúton vagyunk a frame-ek között, akkor 0.5 értéket adunk át a render() függvénynek. Ez alapján az objektumot fél frame-mel előrébb rajzolja ki, 220 pixelnél.

Természetesen előfordulhat, hogy ez az extrapoláció tévesnek bizonyul. A következő frissítés számításakor kiderülhet, hogy a lövedék akadályba ütközött vagy lelassult. Ilyenkor a pozíciót az előző frame és a következő becsült pozíciója között interpoláltuk, de ezt csak a teljes fizikai és AI frissítés után tudjuk meg pontosan.
Az extrapoláció tehát néha hibás lesz. Szerencsére az ilyen korrekciók általában alig észrevehetők – mindenesetre sokkal kevésbé, mint az akadozás, amit akkor kapnánk, ha egyáltalán nem végeznénk extrapolációt.

4. Összefoglalás

A játékciklus a valós idejű interaktív rendszerek, különösen a videójátékok egyik legfontosabb alapvető komponense, amely biztosítja a bemenetek feldolgozásának, az állapotfrissítéseknek és a megjelenítésnek a folyamatos ciklikus működését. A cikk áttekintette a játékhurkok különböző típusait, valamint bemutatta azok működési elveit és gyakorlati alkalmazhatóságát példák segítségével. Az elemzés rávilágított arra, hogy minden megközelítés eltérő előnyökkel és kompromisszumokkal jár: a fix időlépéses hurkok determinisztikus és stabil szimulációt kínálnak, a változó időlépéses módszerek a hardver teljesítményéhez igazodva növelik a reakcióképességet, míg a félig fix megoldások a stabilitás és a hatékonyság közötti egyensúlyt teremthetik meg. A vizsgálatok megerősítették, hogy a játékciklus architektúrájának megválasztása jelentős hatással van a játékmenet folyamatosságára, a fizikai szimuláció pontosságára, valamint a felhasználói élmény minőségére. A cikk kitért továbbá a teljesítményt és a játékélményt javító optimalizálási technikákra is, mint például az eltelt idő simítására, az interpoláció alkalmazására és a maradványkésleltetés kezelésére. E módszerek hozzájárulhatnak a hardverkörnyezetek közötti eltérések kiegyensúlyozásához, valamint a játékok robusztusabb és konzisztens működéséhez. A ciklusok helyes tervezése és implementálása alapvető fontosságú a skálázható, megbízható és magas minőségű videojátékok létrehozásában. A jövőbeli kutatások számára ígéretes irányt jelenthetnek a hibrid megközelítések és fejlett ütemezési technikák, amelyek képesek tovább növelni a valós idejű szimulációk hatékonyságát, miközben megőrzik a játékélmény folyamatosságát és konzisztenciáját, amely a magával ragadó játékmenet alapvető feltétele.

Irodalomjegyzék

[1] Akenine-Möller, T., Haines, E. (2008). *Real-Time Rendering*. 3nd edition. A. K. Peters.

[2] João M. P. Cardoso, José Gabriel F. Coutinho, Pedro C. Diniz (2017). *Embedded Computing for High Performance, Ecient Mapping of Computations Using Customization, Code Transformations and Compilation*. Morgan Kaufmann. 17–56.

[3] Charles Kelly (2012). *Programming 2D Games*. 1st ed. A K Peters/CRC Press.

[4] Jason Gregory (2018). *Game Engine Architecture*. 3 rd ed. A K Peters/CRC Press.

[5] Marín-Lora, C., Chover, M., Rebollo, C., Remolar, I. (2020). A game development environment to make 2D games. *Communication Papers*, Vol. 9 No. 18, 7–23.

[6] Pitt, C. (2016). *The Game Loop, In: Making Games*. Apress, Berkeley, CA. https://doi.org/10.1007/978-1-4842-2493-9_2

[7] Valente, Luis, Aura Conci, and Bruno Feijó (2005). Real time game loop models for single-player computer games. *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*, Vol. 89.

[8] Zamith, Marcelo, Luis Valente, and Esteban Clua (2023). Game loop model properties and characteristics on multi-core cpu and gpu games. *SBGames 2016*.

[9] Gaffer On Games (2023). https://gafferongames.com/post/fix_your_timestep/.

[10] P. Mileff, J. Dudra (2022). Eective Pixel Rendering in Practice. *Production Systems and Information Engineering*, Vol. 10, No. 1, 1–14.

[11] Game Programming Patterns – Game Loop (2023). https://gameprogrammingpatterns.com/game-loop.html