Egy webes játék készítésének programozás-didaktikai szempontjai

Horváth Győző[1], Menyhárt László[2], Zsakó László[3]

{[1]gyozke, [2]menyhart}@inf.elte.hu, [3]zsako@caesar.elte.hu
ELTE IK

Absztrakt. Jelen cikkünket módszertannal foglalkozó szakembereknek és olyan gyakorló pedagógusoknak írtuk, akik az informatika oktatás részeként a programozással is foglalkoznak és szeretnék óráikat különlegessé tenni. Modern böngészők által támogatott HTML5 és JavaScript lehetőségeit mutatjuk be egy egyszerű grafikus játék fejlesztése közben. Ezzel a módszerrel szeretnénk élvezetessé tenni a tanulók számára a módszeres programozást. Az alapok megismertetése után bemutatunk JavaScripthez készült játékkeretrendszereket is, melyekkel még gyorsabban lehet látványos eredményeket elérni, miközben nem mondunk le a programozásról sem.

Kulcsszavak: grafika, programozás, játék, web, HTML5, JavaScript, keretrendszerek.

1. Bevezetés

A webes platform jól használható programozásoktatásra, ahogy azt több példa is mutatja [6,7,8]. Mi is több cikkünkben érintettük már a programozás és a webes környezetek kapcsolatát, mely keretében a JavaScript nyelv használatának lehetőségeit is vizsgáltuk első magas szintű nyelvként [9,10]. Most ezeket a területeket építjük össze és keressük a lehetőséget, hogy a tanulók új ismereteket szerezzenek egy egyszerű, de mégis látványos játék készítésével. A Scratch vagy Blockly grafikus algoritmus reprezentációk után, vagyis a „kis iskola” befejezésekor eljön az idő, amikor a programozás már forráskód írásából is áll. A klasszikus programozás oktatásban matematikai feladatokon keresztül vezetjük be az új ismereteket. Ugyanakkor a diákok érettségi szintje még nem minden esetben éri el az absztrakt és a száraz kitalált adatokhoz erőltetett kérdések feldolgozásához szükséges szintet, így hamar elveszthetik érdeklődésüket. Ráadásul a matematikát szerető diákok köre viszonylag szűk, ezért érdemes lehet más feladatcsoportra építeni.

A középiskolában már elég felkészültek a diákok arra, hogy saját grafikus és képfeldolgozási műveleteket végezzenek, hiszen elemi koordinátageometriai ismereteket megszerzik a matematika órán. Az általunk bemutatott feladat segít az ottani ismeretek jobb elmélyítésében és nem utolsósorban a látványos programozási feladattal az érdeklődésük fenntartásához az informatika iránt.

Cikkünk első felében a módszeres programozás oldaláról közelítünk ötletünk megvalósítása felé. A második részben bemutatjuk a szükséges környezeti és nyelvi alapokat. Végül a játékkeretrendszereket vizsgáljuk meg használhatóság szemszögéből.

2. Módszertani alapok és algoritmusminták

A programozás tanulása hagyományosan egy matematikai/számelméleti feladatkörre szokott épülni, amiben oszthatósággal, prímszámokkal foglalkozunk. Ez megfelelő azon diákoknak, akik a matematika iránt érdeklődnek, szeretik és tudják is a matematikát. Ilyen diákból azonban sajnos nagyon kevés van. Sokan és régóta próbálkoznak azzal, hogy más feladatkört találjanak.

A szerzők egyes esetekben csak a megoldandó feladatok körét változtatják meg, mint Szlávi Péter és Zsakó László biológusoknak írt egyetemi jegyzete [1]. Itt továbbra is a beolvasás, a kiírás, az értékadás az elemi műveletek, amelyeket először elágazással, majd ciklussal szervezünk bonyolultabb programokká.

Egészen más utat követ Seymour Papert [2], aki az elemi utasítások körét változtatta teknőcgrafikai, illetve szövegmanipulációs utasításokra. Az előbbi tanításában a ciklus megelőzi az elágazást, az utóbbiban pedig a rekurzió jelenik meg a ciklusok helyett.

E cikk elméleti alapja az utóbbit követi, változtassuk meg az elemi utasítások körét! Legyen elemi tevékenységünk az értékadás mellett a pont rajzolása képernyőre! [3,4]

Az alapgondolat, hogy egy pontot mozgatunk fix szélességű és magasságú képernyőn. A pontot kezdetben az (o,s) koordinátájú pontra tesszük. Az algoritmusok egyszerűbb megértése miatt most legyen a képernyő mérete 1024 oszlop, 768 sor!

Első feladatként mozgassuk a pontot jobbra 100 egységgel – megjelenik szervezési eszközként a számlálós ciklus!

```
Előre(N):
  s:=200; o:=200
  Ciklus N-szer
    Pont(o,s); o:=o+1
  Ciklus vége
Eljárás vége.
```

Próbáljuk ki a kép jobb széléig mozgatást – megjelenik szervezési eszközként a feltételes ciklus!

```
Széléig(s,o):
  Ciklus amíg o<1024
    Pont(o,s); o:=o+1
  Ciklus vége
Eljárás vége.
```

Megfigyelhetjük, hogy mindkét mozgatással egy vonalat húztunk. A pont rajzolására van azonban egy olyan lehetőségünk, hogy a pontot színe szerinti kizáró vagy művelettel tegyük ki a képernyőre. A kizáró vagy művelet tulajdonsága, hogy valamire kétszer alkalmazva, visszakapjuk az eredetit, azaz `A XOR B XOR B=A`. Ekkor az eddigi rajzolásról áttérünk az animációra.

A következő eljárásban – egy lassítást helyezve a programba – láthatjuk a mozgó pontot:

```
Széléig(s,o):
  Ciklus amíg o<1024
    Pont(o,s); o:=o+1; Pont(o,s); Várakozás
  Ciklus vége
Eljárás vége.
```

Adjunk a pontnak egy tetszőleges irányt (`do`,`ds`), ebbe az irányba mozgassuk a pontot a képernyő széléig – kiválasztás programozási tétel (hiszen megadjuk azt a képernyő koordinátát, ahol a pont meg fog állni!

```
Széléig(s,o,ds,do):
  Ciklus amíg o>0 és o<1024 és s>0 és s<768
    Pont(o,s); o:=o+do; s:=s+ds; Pont(o,s); Várakozás
  Ciklus vége
Eljárás vége.
```

Ha a képernyőre korábban már rajzoltunk vonalakat (ún. falakat), akkor is álljunk meg, ha falhoz értünk – keresés programozási tétel a képernyő pontjai színe érzékelésével!

```
Széléig vagy falig(s,o,ds,do):
  Ciklus amíg o>0 és o<1024 és s>0 és s<768 és pontszín(o,s)≠háttér
    Pont(o,s); o:=o+do; s:=s+ds; Pont(o,s); Várakozás
  Ciklus vége
Eljárás vége.
```

A képernyő széléről a pontunk szabályosan verődjön vissza (és végtelen sokáig menjen) – megjelenik az elágazás!

```
Visszaverődve(s,o,ds,do):
  Ciklus
    Pont(o,s); o:=o+do; s:=s+ds; Pont(o,s); Várakozás
    Ha o>1023 akkor o:=1023-(o-1023); do:=-do
    Ha o<0 akkor o:=-o; do:=-do
    Ha s>767 akkor s:=767-(s-767); ds:=-ds
    Ha s<0 akkor s:=-s; ds:=-ds
  Ciklus vége
Eljárás vége.
```

A képernyő széléről a pontunk szabályosan verődjön vissza, de ha belső falat talál el, akkor álljon meg!

```
Visszaverődve falig(s,o,ds,do):
  Ciklus pontszín(o,s)≠háttér
    Pont(o,s); o:=o+do; s:=s+ds; Pont(o,s); Várakozás
    Ha o>1023 akkor o:=1023-(o-1023); do:=-do
    Ha o<0 akkor o:=-o; do:=-do
    Ha s>767 akkor s:=767-(s-767); ds:=-ds
   Ha s<0 akkor s:=-s; ds:=-ds
  Ciklus vége
Eljárás vége.
```

A képernyő széléről a pontunk szabályosan verődjön vissza, a képernyőn haladás közben fokozatosan lassuljon! A lassulás megoldása: az időegységenkénti elmozdulás (azaz a sebesség) minden lépésben c-szeresre változzon! Ha c<1, akkor lesz lassulás, c>1 esetben pedig gyorsulás lesz.

```
Visszaverődve lassulva(s,o,ds,do,c):
  Ciklus
    Pont(o,s); o:=o+do; s:=s+ds; Pont(o,s); Várakozás
    Ha o>1023 akkor o:=1023-(o-1023); do:=-do
    Ha o<0 akkor o:=-o; do:=-do
    Ha s>767 akkor s:=767-(s-767); ds:=-ds
    Ha s<0 akkor s:=-s; ds:=-ds
    ds:=c*ds; do:=c*do
  Ciklus vége
Eljárás vége.
```

A képernyő széléről a pontunk szabályosan verődjön vissza, lefelé haladva gyorsuljon a mozgása, felfelé haladva pedig lassuljon – fizikai tudás beépítése a programba! Mivel a képernyő egyik koordinátája nem a szokásos koordinátarendszer irányban növekszik, ezért a lefelé itt képernyő sorindex növekedést jelent.

```
Visszaverődve gyorsulva-lassulva(s,o,ds,do,c):
  Ciklus
    Pont(o,s); o:=o+do; s:=s+ds; Pont(o,s); Várakozás
    Ha o>1023 akkor o:=1023-(o-1023); do:=-do
    Ha o<0 akkor o:=-o; do:=-do
    Ha s>767 akkor s:=767-(s-767); ds:=-ds
    Ha s<0 akkor s:=-s; ds:=-ds
    Ha ds≥0 akkor ds:=ds+1 különben ds:=ds-1
  Ciklus vége
Eljárás vége.
```

Megfelelő feladatokat kapunk az algoritmikus struktúrák kialakítására. Látványos feladatok, játékokká fejleszthetők. Kialakíthatók bennük akár a programozási tételek is – megszámolás (falakon áthaladás vagy szélekről visszaverődés számlálása), kiválogatás (az elért falak átszínezése).

A fenti algoritmusok egyszerű, hagyományos programozási nyelven is megvalósíthatók – átvezetésnek használhatók is –, de messze nem olyan élvezetesek, látványosak, mint amit modern eszközökkel, modern intelligens felületeken el lehet érni. A továbbiakban ezeknek az algoritmus mintáknak a segítségével igyekszünk elkészíteni a közismert téglafalbontó játékot böngészőben futtatva HTML5 és JavaScript felhasználásával.

3. Alapok és a környezet bemutatása

A diákok legalább annyira ismerik a böngészőt, hogy azt használják. Ezt a környezetet szeretnénk mi felhasználni, hogy fordítóprogramok és futtató környezet telepítése nélkül lehetőleg platform-független módon, már forráskódot írva, de grafikus és így látványos alkalmazásokat készíthessenek. A HTML5 megjelenésével lehetőség nyílik a grafikus elemek natív használatára a web oldalakon a `canvas` elem segítségével, amit már minden elterjedt böngésző támogat.

A `canvas` lehetőséget ad kétdimenziós („`2d`”) és háromdimenziós („`webgl`”) kontextus kezelésére. Mi most a `2d`-t használjuk, vagyis egy képet hozunk létre, módosítunk, használunk. A következő példakódban egy fekete hátterű kép (téglalap) közepére kirajzolunk egy sárga színnel kitöltött kört. Ehhez szükségünk van a `canvas` tag-re a `body`-ban, egy JavaScript függvényre (`main`), ami az oldal betöltődésekor lefut és le van benne kódolva a canvas 2d-s kontextusának megszerzése és a rajzoló függvény (`draw`) meghívása. A konkrét téglalapot rajzoló függvény a `fillRect`, míg a kört és kitöltését két külön paranccsal kell megtenni (`arc`, `fill`). Ezek részletes leírását ebben a cikkben nem részletezzük, hiszen több weboldalon is megtalálhatóak a leírások, pl [11]-ben.

```
<!DOCTYPE html><title>Example</title>
<canvas id="canvas" width="1024" height="768"></canvas>
<script type="text/javascript">
  var canvas = document.getElementById('canvas');
  var ctx = canvas.getContext('2d');
  function draw() {
    //Clear
    ctx.fillStyle = 'black';
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    //Ball
    ctx.fillStyle = 'yellow';
    ctx.beginPath();
    ctx.arc(512, 384, 50, 0, Math.PI*2, true);
```

```
      ctx.closePath();
      ctx.fill();
    }
    draw();
  </script>
```

Amennyiben mozgásokat is szeretnénk megjeleníteni, akkor időnként újra kell rajzolnunk a képet. Az előző fejezetben említettük, hogy az objektum újra kirajzolása nem elég, mert akkor az előző állapot is a képen marad. Az `XOR` függvény hatékony egyszerű, kétféle színű ábráknál, de a következő példában inkább az általánosabb és legegyszerűbb, az egész képet letörlő és újrarajzoló megoldást használjuk.

Ez a rajzolás elég erőforrás igényes, ezért nem csak egyszerűen időközönként rajzoltatjuk ki az új képet a `setInterval` függvény segítségével. Nem a következő rajzolás idejét határozzuk meg az előzőnél a `setTimeout` függvény segítségével, hanem felhasználjuk a böngészők teljesítményét jobban figyelő `requestAnimationFrame` függvényt [12], ami nagyon hasonlít a `setTimeout`-ra, de a böngészőt kéri meg, hogy a mi következő kirajzolásunkat (`gameloop`) végző függvény hívását időzítse figyelembe véve a böngésző képfrissítését. Ezért előfordulhat, hogy különböznek az időintervallumok a kirajzolások között, vagyis nekünk kell kiszámolnunk a két kirajzolás között eltelt időt, majd abból frissítjük az objektum új pozícióját (`update`) és kirajzoljuk az új helyre (`draw`).

Mivel a fenti algoritmusok kihasználták az azonos időközöket, ezért a technológia megkötései miatt kicsit át kell írnunk azokat a kódolás során. A `main` függvény tehát úgy módosul, hogy létrehozunk egy `body` objektumot a labdának és egy `prevT` nevű változót az időpont elmentéséhez, aminek a segítségével az eltelt időt ki tudjuk számolni. Az `update` függvény pedig a labda új pozíciójának kiszámítását végzi az eltelt idő alapján. A `draw` függvényben pedig felhasználjuk a `body` aktuális értékeit a rajzoláshoz.

```
var body = {
  x: 512,  y: 384, r: 50,  // px
  vx: 100, vy: 60 // px/s
};
var prevT = Date.now();
function gameloop() {
  var t = Date.now();
  var dt = t - prevT;
  prevT = t;
  window.requestAnimationFrame(gameloop);
  update(dt);
  draw();
}
function update(dt) {
  body.x += body.vx * dt/1000;
  body.y += body.vy * dt/1000;
  var frame = {
    bfx: body.r, jax: canvas.width - body.r,
    bfy: body.r, jay: canvas.height - body.r
  }
  if (body.x > frame.jax) {
    body.x = frame.jax - (body.x - frame.jax);
    body.vx = -body.vx;
  }
  if (body.x < frame.bfx) {
```

```
      body.x = frame.bfx + (frame.bfx - body.x);
      body.vx = -body.vx;
    }
    if (body.y > frame.jay) {
      body.y = frame.jay-(body.y- frame.jay);
      body.vy = -body.vy;
    }
    if (body.y < frame.bfy) {
      body.y = frame.bfy + (frame.bfy - body.y);
      body.vy = -body.vy;
    }
    body.vy += 1;
  }
  function draw() {
    //Clear
    ctx.fillStyle = 'black';
    ctx.fillRect(0, 0, canvas.width, canvas.height);
    //Ball
    ctx.fillStyle = 'yellow';
    ctx.beginPath();
    ctx.arc(body.x, body.y, body.r, 0, Math.PI*2, true);
    ctx.closePath();
    ctx.fill();
  }
  gameloop();
```

Innentől kezdve a közismert téglafalbontó játék első, egyszerű verziójának elkészítése már nagyon egyszerű, hiszen az eddig használt elemeket kell használni. A labda objektum (`body`) mellett lesz még a tégla objektumok listája (`bricks`). Az `update` függvényt úgy kell átírni, hogy ne csak a széleknél verődjön vissza, hanem az eltalált tégláknál is, de azokat az első találatkor törölni kell a listából is. A `draw` függvény pedig rajzoljon ki minden objektumot a listából.

A következő lépésben még egy ütköző (`bumper`) objektumot kell felvenni a méreteivel és koordinátáival. Erről ugyancsak visszaverődik a labda, de nem tűnik el és vízszintesen mozgatni tudjuk, vagyis a pozícióját billentyűzettel vagy egér mozgatásával tudjuk vezérelni. Itt újabb programozás-technikai fogalommal találkozunk, az esemény-vezérléssel. A billentyűzeten lenyomott bármelyik gomb hatása egy esemény, melyet a böngésző figyel és amennyiben saját függvényt (`moving`) rendeltünk hozzá (`addEventListener`), akkor az lefut. Így tudjuk a balra illetve jobbra nyilak lenyomásakor a mozgó objektum koordinátáját változtatni a megfelelő módon. Ehhez csak a következő bővítést kell hozzáadnunk a JavaScript-ünkhöz.

```
var bumper = {
  x: 512,  y: 758, w: 100,  h: 20,  // px
  moving: 25 // px
}
function moving(event) {
  if (event.keyCode==37) {
    //moving left
    if (bumper.x-bumper.moving-(bumper.w/2)>0) {
      bumper.x=bumper.x-bumper.moving;
    } else {
      bumper.x=bumper.w/2;
    }
  } else if (event.keyCode==39) {
```

```
      //moving right
      if (bumper.x+bumper.moving+(bumper.w/2)<canvas.width){
        bumper.x=bumper.x+bumper.moving;
      } else {
        bumper.x=canvas.width-(bumper.w/2);
      }
    }
  }
  document.addEventListener("keydown",moving,true);
```

Ezeken a módosításokon kívül még a visszaverődéseket kell az `update` függvényben lekezelni és el is készült a sajátgyártmányú játékunk.

4. Játékkeretrendszerek használata

Az előző fejezetben láthattuk, hogy a lényegi logikát tartalmazó `update` függvény mellett számos további kódrészlet gondoskodott arról, hogy alkalmazásunk a böngészőben megjelenjen és működjön. Ezek a kódrészletek szükségszerűek, hiszen nélkülük a program futtatása, illetve megjelenítése nem valósulna meg, mégis az adatleíró és a logikát jelentő kódrészhez képest egyszerűbb, kevesebb kreativitást és gondolkodást igénylő, sematikus, minden programban hasonló módon megjelenő programkódok ezek. A tanítás során érdemes ezekre felhívni a figyelmet, és ahol lehet funkcionalitásukat paraméteres függvényekben újrahasznosítani. Ezzel már egy lépést tettünk előbbre az absztrakciós skálán, hiszen az egybefüggő kódhalmazból képesek voltunk kiemelni olyan részeket, amelyek egységet alkotnak és alprogramokba szervezhetőek.

A következő lépés ezen ismétlődő és újrahasznosítható kódrészleteknek egy egységes környezetbe ágyazása, mely környezet ezáltal a fejlesztés keretét biztosítja. Ezzel elérkeztünk a következő absztrakciós szintig, ahol már előre definiált elképzelés szerint és előre megadott segédfüggvények segítségével írhatjuk meg a megoldásunkat. Ezeket a környezeteket keretrendszereknek hívjuk. A keretrendszereknek alapvetően kettős feladatuk van: egyrészt a fejlesztés menetéről, a kódrészletek helyéről, a működés filozófiájáról határozott elképzeléssel rendelkeznek, másrészt a gyakran visszatérő problémákra jól bevált gyakorlatok szerinti, előre elkészített megoldást szolgáltatnak. Koncepció és eszközkészlet kettőse jellemzi őket.

Webes játékfejlesztés kapcsán a legtöbb keretrendszer kész megoldást nyújt a játékciklus megszervezésére, két ciklus között eltelt idő számítására, a megjelenítés technológiájára és módjára, a játékban szereplő objektumok definiálására, sokszor ezek kölcsönhatásának vizsgálatára, a különböző erőforrások (kép, hang, animációk, stb.) kezelésére és még sorolhatnánk a jellemzőket, amelyek egyébként keretrendszerről keretrendszerre változnak. Természetesen mindegyik keretrendszernek sajátos megoldása van a kód szervezésére: némelyik abszolút moduláris, mások monolitikusabbak, vannak deklaratívabb és imperatívabb megközelítéseket alkalmazók.

A sok egyéb jellemző közül oktatási szempontból külön kiemelendők a játékban szereplő entitások kezelése. Noha a naiv megoldásban is szükséges volt a pont pozíciójának nyilvántartása, és ennek módosítása, ezek funkcionális egységéről programnyelvi eszközzel nem gondoskodtunk. A játékban szereplő objektumok elemzésével az adatábrázolás és még közelebbről az objektum-orientált paradigma jelenik meg viszonylag intuitívan a feladatmegoldás során. Ez is egy absztrakciós lépés, amely során azonosítjuk az elképzeléseinkben szereplő játéktér objektumait, átgondoljuk, hogy milyen adatok és műveletek jellemzik őket. Ezt természetesen a naiv megoldásban is megtehettük volna, a játék keretrendszerek azonban legtöbbször eleve ilyen módon biztosítják az állapottér részekre bontását.

A konkrét megvalósítás természetesen a használt keretrendszertől függ. E cikkben az ez irányú továbblépési lehetőséget a Frozen nevű játék-keretrendszerrel [5] demonstráljuk, mivel ebben még viszonylag jól felismerhető a naiv kiindulási pont, ugyanakkor a fentebb felsorolt absztrakciók is könnyen alkalmazhatóak. Az előkészületek, a konfigurálás és a játékciklus megszervezését a `GameCore` objektum vállalja magára, amelyet a felparaméterezés és példányosítás után a `run` metódussal indíthatunk:

```
var game = new GameCore({
  canvasId:         'canvas',
  gameAreaId:       'gameArea',
  canvasPercentage: 0.95,
  update:           update,
  draw:             draw
});
game.run();
```

A játékban pattogó labdát egy külön modulban írjuk, ahol egységbe zárva találhatók meg a labdát jellemző adatok és az ahhoz tartozó műveletek (pl. `update` metódus). Ugyancsak itt lehet definiálni azt is, hogy a labda hogyan jelenjen meg a rajzvásznon (`draw` metódus):

```
function Ball() {
  this.x = 50; this.y = 20; this.r = 5;
  this.vx = 100; this.vy = 60;

  this.update = function (millis) {
    //Korábbihoz hasonló update függvény
  };
  this.draw = function (ctx) {
    //Korábbihoz hasonló kirajzolás
  };
}
```

Ezzel együtt a játékciklus `update` és `draw` metódusa a labdaobjektum `update` és `draw` metódusának hívásából áll csupán.

Az elemek mozgatása és kölcsönhatása sokszor a valós világ törvényszerűségeit követi. Ilyen játékok esetében ezeknek a törvényszerűségeknek a kivitelezését ugyancsak rábízhatjuk egy absztrakt rétegre: ezek a fizikai keretrendszerek. A fizikai motorok ugyancsak objektumokkal dolgoznak, de a pozíció és sebesség mellett további, valós világbeli tulajdonságokkal vannak felruházva, tömeggel, impulzussal, perdülettel, alakkal és anyaggal, és ezek az objektumok a gravitációs térben mozogva kölcsönhathatnak, ütközhetnek egymással. A fizikai keretrendszerek sokszor paraméterezési kérdéssé egyszerűsítik az objektumok létrehozását és mozgatását: nem a viselkedés megadása adja a kód nagyobbik részét, hanem világ felparaméterezése és benépesítése.

A sokféle fizikai motor közül (p2 [13], `box2d` [14]), a Frozen keretrendszer a `box2d` fizikai motorral biztosít nagyobb integrációt a `BoxGame` osztályon keresztül: az ehhez hozzáadott `box2d` objektumok automatikus mozgatását és kirajzolását a keretrendszer végzi. Az alábbi példakódban látható, hogy a fentebb megadott `GameCore` osztály kiegészül a `box` paraméterrel, labdát pedig az `addNewBall` metódussal lehet hozzáadni:

```
var game = new BoxGame({
  //Korábbi paraméterek
  //...
  box: new Box({gravityY: 0}),
  addNewBall: function () {
    var ball = new Ball();
```

```
      game.addBody(ball);
      game.box.applyImpulseDegrees(ball.id,155,ball.impulse*0.75);
    }
});
game.addNewBall();
game.run();
```

A fizikai világhoz már csak a működést biztosító objektumokat kell hozzáadni. Definiálni kell a négy falat, valamint a labdát. A labda osztály a `box2d` függvénykönyvtár `Circle` osztályából származva örökli a kör alakú fizikai testek tulajdonságait (egy `dcl` nevű függvénykönyvtár segítségével). Mozgatását és kirajzolását ugyancsak a keretrendszer biztosítja, a kirajzolás természetesen testre szabható. Az alábbiakban a tökéletesen rugalmasan visszapattanó labdát szimuláljuk.

```
var Ball = dcl(Circle, {
  ball: true,
  x: 60, y: 210, radius: 10,
  restitution: 1.0, friction: 0, impulse: 5,
  constructor: function(){
    this.id = this.id || _.uniqueId();
  }
});
```

Ha a végső megoldást nézzük, akkor ezen a szinten eltűntek azok a programozási tételek, amelyeknek gyakoroltatásához éppen a játékokat vettük elő, mint érdekes feladatokat. Az ottani sorozatokat és ciklusokat elvont programozói felületek mögé rejtettük. Természetesen a programozási tételek itt is megjelenhetnek, csak már ennek az absztraktabb szintnek az objektumaival. A programozási tételekhez sorozatok kellenek. A játékoknak ezen absztrakciós szintjén fizikai testekkel, entitásokkal, sprite-okkal dolgozunk. A játéklogika megírásához azonban szükséges ezek vizsgálata, szükséges az ezeken történő végighaladás és szükségszerűen előjönnek eközben a programozási tételek is.

5. Konklúzió

A cikkben bemutatott környezet és technológiák lehetőséget adnak arra, hogy látványos és játékos alkalmazásokat készítsünk, melyekkel motiválhatjuk az iskolás gyerekeket a programozás tanulásra. Az itt bemutatott út az algoritmikus gondolkodástól kezdve a technológiai ismereteken keresztül egy magasabb absztrakciós szintig vezeti el az olvasót és reméljük lesz más vállalkozó pedagógus is, aki kipróbálja e módszert a tanulóival.

Úgy gondoljuk, hogy a programozás megszerettetésére több út van, egyik sem lehet kizárólagos. Azt gondoljuk, hogy egy jelentős rétegnél sikeres lehet a webes játékfejlesztésen alapuló módszer – több diáktól is hallottuk, hogy a számítógépes játékoknál is jobb dolog játékprogramot készíteni.

A cikk tervezése közben szempont volt, hogy a módszeres programozást vegyük alapul, vagyis az algoritmus-mintákra építsünk, motiváljuk a diákokat az ismert webes környezet és a játék hívószavakkal. Haladunk előre a kódolással és közben új technológiát és programozástechnikát vezetünk be. Végül a keretrendszer bevezetésével absztrakciós szintet lépünk.

A cikkben bemutatott programozási feladattal tehát teljes választ adtunk a didaktika által meghatározott cél, tartalom, folyamat, szervezés, módszer, eszköz kritériumokra.

Irodalom

1. P. Szlávi, L. Zsakó: Bevezetés a számítástechnikába. Egyetemi jegyzet. Tankönyvkiadó. (1987)
2. S. Papert: Észrengés. A gyermeki gondolkodás titkos útjai. SZÁMALK. (1988)
3. L. Zsakó, P. Szlávi: Grafikai alapalgoritmusok. INF.O.'94 Informatika és számítástechnika tanárok konferenciája. Békéscsaba, Magyarország, 1994.11.17-1994.11.19. (1994)
4. L. Zsakó, P. Szlávi: Az informatika oktatása. Budapest, ELTE Informatikai Kar. (2014)
5. http://frozenjs.com/docs/ (utoljára megtekintve: 2015.10.31.)
6. https://code.org/learn (utoljára megtekintve: 2015.10.31.)
7. https://blockly-games.appspot.com/ (utoljára megtekintve: 2015.10.31.)
8. https://www.khanacademy.org/computing/computer-programming (utoljára megtekintve: 2015.10.31.)
9. Gy. Horváth, L. G. Menyhárt: Teaching introductory programming with JavaScript in higher education (Volume 1. pp. 339-350) doi: 10.14794/ICAI.9.2014.1.339 (2014)
10. Gy. Horváth, L. G. Menyhárt: Oktatási környezetek vizsgálata a programozás tanításához. INFODIDACT 2014, Zamárdi, 2014.11.23-24. http://infoera.hu/infoera2014/ea/infodidact2014_HorvathGy_MenyhartL.pdf (2014)
11. https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API (utoljára megtekintve: 2015.10.31.)
12. https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame (utoljára megtekintve: 2015.10.31.)
13. https://schteppe.github.io/p2.js/ (utoljára megtekintve: 2015.10.31.)
14. http://box2d-js.sourceforge.net/ (utoljára megtekintve: 2015.10.31.)