

Valósídejű programozás Processing környezetben

Dr. Falucskai János

2018



Valós idejű programozás Processing környezetben PDF fájlformátumban megjelent elektronikus tananyag

Szerző: Dr. Falucskai János, főiskolai docens

Nyíregyházi Egyetem

Matematika és Informatika Intézet

Készült: 2018. szeptember 30.

Lektorálta: Eichinger László, főiskolai mérnöktanár

Készült az alábbi pályázati projekt támogatásával:

EFOP-3.5.1-16-2017-00017 „NYE-DUÁL- Új utakon a duális felsőoktatással a Nyíregyházi Egyetemen, az Északkelet-Magyarországi térség felemelkedéséért”



Szerzői jogok: Jelen tananyag a Creative Commons: Nevezd meg! – Így add tovább! 4.0 Nemzetközi Licenc (CC BY-SA 4.0) feltételeinek megfelelően szabadon felhasználható.

Tartalomjegyzék

1. Bevezetés	4
2. Valós idejű programozás	5
2.1. Valós idejű programozás fogalma	5
2.2. Valós idejű rendszerek és feladatok osztályozása	5
2.3. RT architektúrák	8
3. A Processing programozási nyelv	10
3.1. A Processing programozási nyelv alapjai	11
3.1.1. A nyelv alapelemei	12
3.1.2. Típusok	13
3.1.3. Operátorok	15
3.1.4. Utasítások	16
3.1.5. Függvények, vagy más néven metódusok	21
3.1.6. A setup() függvény	22
3.1.7. A draw() függvény	23
3.2. Az IDE	24
3.3. Egy egyszerű feladat megoldása	25
4. Egyszerű időfüggő feladatok kódolása Processing környezetben	29
4.1. Digitális óra	29
4.2. Analóg óra	30
4.3. Fizikai feladatok modellezése	37
4.3.1. Ütközés szimulálás	37
4.3.2. Egyenletes körmozgás és harmonikus rezgőmozgás	38
4.3.3. Ferde hajítás szimulációja	40
4.4. Időalapú fraktál	42
4.5. Fényűjság	48
5. Összetett időfüggő feladatok kódolása Processing környezetben	51
5.1. Billentyűzet kezelése	54
5.1.1. Szám tippelés	55
5.2. Az egér kezelése	58
5.2.1. Négyzet vonszolása	59
5.2.2. Szakasz rajzolása	60
5.3. Szálkezelési lehetőségek, szinkronizálás	63
6. Összefoglalás	67

1. Bevezetés

A jegyzetnek több célja van, szeretnénk megmutatni egy univerzális lehetőséget az érdeklődőknek. A felhasználási területekben közös a kevés időráfordítás, azaz rövid forráskódú programokkal tudunk megoldani összetettebbnek tűnő feladatokat minimális Java ismeretekre alapozva. A tananyag elsajátítása után képesek lesznek

- Időfüggő feladatok megírására.
- Különbözőbb előkészületek nélkül grafikai felületű programok létrehozására.
- Az egér- és billentyűzetesemények egyszerű kezelésére.
- Egy könnyen tanulható programozási nyelven programok írására.

A jegyzet hasznos segédanyagként fog szolgálni azok számára is, akik a "Valósídejű programozás" kurzust felvették.

2. Valós idejű programozás

A programozás lényege, hogy kívülről működtetjük a számítógépet: az elvégzendő tevékenység leírása a program, tehát feladatmegoldást adunk meg. Valós idejű programozás esetén tekintettel kell lennünk az időre, mely alatt a "valós" időt értjük. Valós idő esetünkben a tényleges idő. Adott idő alatt lefutó programok, adott ideig történő cselekvéseket végző alkalmazásokat szeretnénk írni. Az egyik legkézenfekvőbb feladat a pontos idő kiírása, de ide tartoznak azok a feladatok is, amelyek nem az aktuális időhöz kötődnek, hanem adott idő alatt hajtódnak végre, avagy más folyamatokkal időbeli kapcsolódásuk van. A programozás területén ezek a megoldandó feladatok új problémákat vetnek fel, avagy a már megoldott, felvetődött problémák új osztályba történő besorolást nyernek.

2.1. Valós idejű programozás fogalma

A fentiek alapján megadható a valós idejű programozás fogalma: Valós idejű feladat megoldására irányuló program, ahol a valós idejű feladat valós időtartományban számítógéppel megoldandó feladat.

Általában a konkrét feladatnak ez csak egy kisebb részét jelenti, de a következőkben csupán erre a speciális részre fogunk koncentrálni, a megoldandó feladatokban a hangsúly mindig az időbeliségen lesz majd. Az alábbi esetei vannak a valós idejű problémáknak időbeliségi szempontok alapján:

- A feladat végrehajtásának van felső időkorlátja (maximum adott idő alatt be kell fejeződnie)
- A feladat végrehajtásának van alsó időkorlátja (minimum adott ideig kell tartania)
- A feladat végrehajtása során figyelembe kell venni külső eseményeket, s azokra lehető leghamarabb reagálni kell (pl. egér és billentyűzet események)
- A feladat végrehajtása során figyelembe kell venni több folyamat eseményeit, azokat szinkronizálni kell

Összefoglalva megállapítható, hogy egyrészt a tényleges idővel, másrészt pedig a program során létrejövő folyamatok végrehajtási idejével kapcsolatos feladatokat kell megoldani. A tananyag természetesen nem csak elméleti jellegű részeket tartalmaz, a továbbiakban megadásra kerülnek a gyakorlati alkalmazások lehetőségei, konkrét programozási nyelven konkrét feladatok megoldásaira mutatunk példákat, valamint utalunk a feladatokra épülő egyéb problémák felvetésére. Talán felesleges hangsúlyozni, de az algoritmusok és forráskódok megértése nem helyettesíti azok megírását. Erősen javallott az implementálásuk, programírás közben lehet a legtöbbet tanulni, a hibák keresése és javítása adja a gyakorlatot.

2.2. Valós idejű rendszerek és feladatok osztályozása

Alapvetően a felhasználás területe szerint lehet besorolni a valós idejű programozási feladatokat. Az alábbi osztályozás igyekszik megvalósítani egy lehetséges felosztást, de természetesen ezen kívül egyéb felhasználási lehetőségek is lehetségesek, erre utal minden esetben a "..." menüpont.

1. Ipari irányítási feladatok

- (a) Gyártósorok
- (b) Robotok
- (c) Repülés irányítás
- (d) ...

2. Kereskedelmi irányítási feladatok

- (a) Bank
- (b) Tőzsde
- (c) Biztonsági rendszerek
- (d) ...

3. Társadalom irányítási feladatok

- (a) Szavazás
- (b) Okmányiroda
- (c) Földhivatal
- (d) ...

4. Számítástechnika

- (a) Multiprocesszoros rendszerek
- (b) Multitasking-os rendszerek
- (c) Multiuser-es rendszerek
- (d) Időfüggő feladatok
- (e) ...

A továbbiakban a számítástechnika keretein belül fogunk megállapításokat tenni. Megadjuk a valósidejű rendszer fogalmát:

A valósidejű rendszer a fizikai folyamat lezajlásával egy időben végzi el az

- Információ szerzés (mérés)
- Információ feldolgozás
- Beavatkozás

tevékenységét úgy, hogy a folyamat ellenőrizhető és irányítható legyen.

Tehát a valósidejű programozás –ahogy fentebb is említésre került– időkorlátos feladatok megoldására szolgál. Az időkorlát nem mindig konkrét, pl. szinkronizálások esetén nem lehet kötni adott időpillanathoz, vagy adott időtartamhoz.

Nagyobb munkákban előforduló valósidejű programozást tartalmazó projekt esetén folyamat-irányító szoftverről beszélünk. Ebben az esetben projekt alapú tervezés történik, ugyanis a feladat teljes megoldása a projekt.

A projekt moduljai:

- Központi szint: alkalmazások indítása, beléptetés

- Operátori interface: folyamatok megjelenítése, vezérlése
- Grafikus tervező eszközök: operátori interface elkészítésére szolgál
- Parancsvégrehajtó: a szoftver nyelvén program írható
- Könyvtárkezelő: rutinok háttértára
- Alarmkezelő
- Adatgyűjtő
- Gateway
- Grafikonkezelő
- Stb...

A projekt végrehajtása során a következő feladatok kerülnek napirendre:

- Probléma elemzése
- Tervezés
- Rendszer specifikáció
- Implementálás
- Tesztelés
- Próbaüzem
- Karbantartás
- Dokumentáció készítés

Nem könnyű feladat a projekt tervezése, általában a következő problémákkal lehet szembesülni:

- Mit kell megvalósítani?
- Reális feladatmeghatározás
- Gazdaságos megoldás
- Programozó kiválasztása
- A tesztelés során a vevőnek a jószág bizonyítása
- Dokumentálás, jogvédelem

2.3. RT architektúrák

A valós idejű rendszer a nemzetközi szakirodalomban az RTS (RealTime System–RTS) néven kerül említésre. Alapvetően több osztályozása ismert:

- Szigorúan (hard) RTS: Az időkorlát átlépése végzetes hiba, semmilyen körülmények között sem engedhető meg, mivel nagy anyagi kárt, és/vagy emberéletet követelhet.
- Lazán (soft) RTS: Az időkorlát átlépése kezelendő, mivel lényeges problémát okozhat, mely zavarja a rendszer működését, rontja a felhasználói élményt.
- Nem RTS: Az időkorlát átlépése nem érdekes, nem kell foglalkozni a folyamat végrehajtódásának időtartamával.

További fogalmak:

- Válaszidő: Az eseményre való reagálás ideje.
- Esemény: A rendszerre hatással bíró input.
- Rendszeridő: A válaszidők maximuma.

A valós idejű programozás nagy erőforrást igényel, ezért elterjedt megoldásként használják a multiprocesszoros rendszereket, ahol ugyanazon az algoritmuson több mikroprocesszor dolgozik egymással párhuzamosan. Megkülönböztetünk úgynevezett lazán és szorosan csatolt rendszereket. Lazán csatolt rendszerek esetében csatornákon folyik a kommunikáció, más-más operációs rendszer alattiak a mikroprocesszorok, míg szorosan csatolt rendszerénél közös erőforrásokat használnak, közvetlen a kommunikáció pl. memórián keresztül és természetesen azonos az operációs rendszer.

- Dinamikusnak nevezzük egy ilyen architektúrát, ha több azonos típusú mikroprocesszort tartalmaz, ebben az esetben bonyolult a terhelést elosztó szoftver, viszont könnyű, és olcsó a javítás az egyforma hardverelemek miatt, melyeknek közel azonos a terhelése.
- Statikus az architektúra, ha a mikroprocesszorok különbözők, ebben az esetben a terhelést elosztó szoftver egyszerűbb, de nehéz és drága javítás a processzorok eltérései miatt, valamint nem egyenletes a terhelésük.

Valós idejű program tervezésénél a megszokott lépéseket követjük a fogalmi modellből az implementációs modell kialakítása során:

- Az absztrakció során elválasztjuk a lényeges információkat a lényegtelenektől
- A dekompozíció végrehajtásával megtörténik a probléma részfeladatokká bontása
- Nagyobb rendszerek esetén megszületik a formalizálás, tehát rögzítésre kerül a leírás valamilyen módszerrel
- Modellezzük, majd következik a létrehozott modell helyességének ellenőrzése
- Implementáljuk, azaz megvalósítjuk a végleges modellt.

- Utolsó előtti lépésként történik meg a validálás, a specifikációnak való megfelelés ellenőrzése: "Are we building the right product?", azaz valóban azt teszi-e az alkalmazás, amit a megrendelő kért.
- Végül verifikációval fejezzük be a fejlesztés ciklusát: "Are we building the product right?", azaz jól írtuk-e meg a programot, vagy lehetett-e volna optimálisabban, biztonságosabban.

Az elméleti rész végén felsorolásra kerülnek a valós idejű programokban előforduló időzítési követelmények:

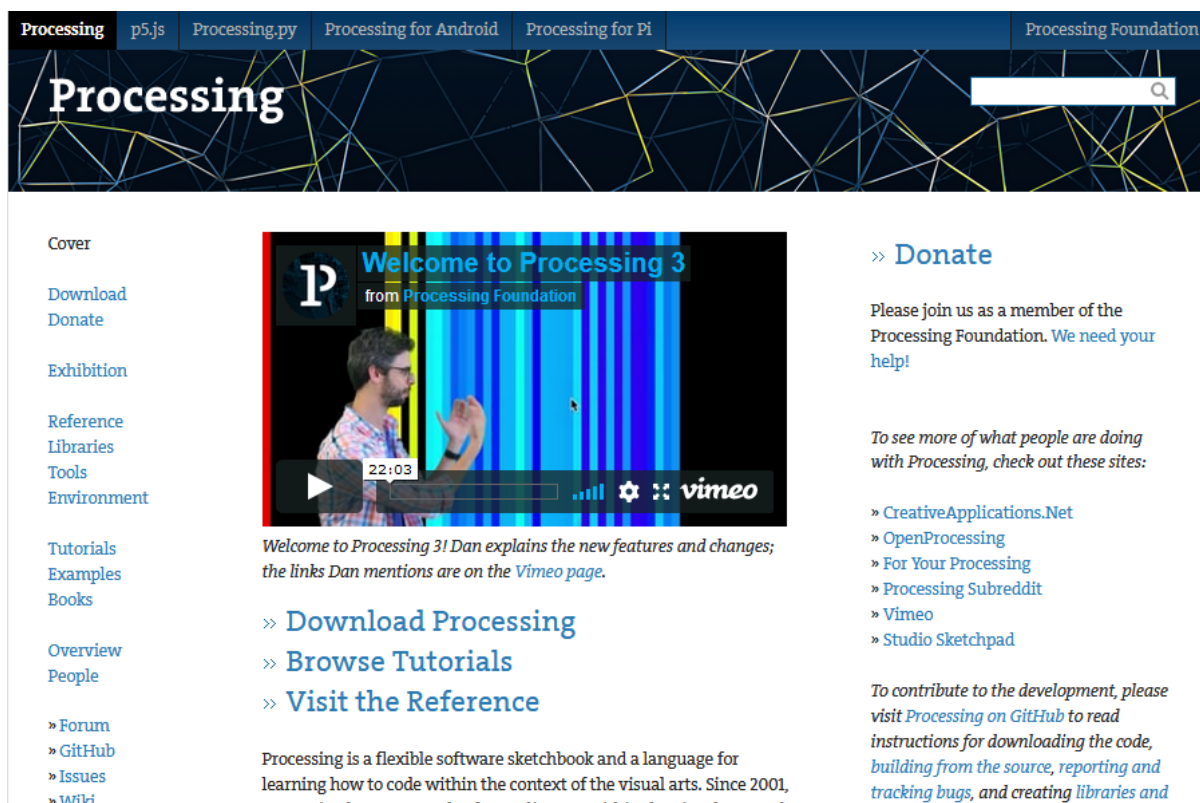
- periodikus, ha adott időközönként kell megismételni egy eseményt.
- határidős, amennyiben megadjuk az esemény bekövetkezésének maximumát, vagy minimumát, esetleg mindkettőt.
- időzített, ekkor pontos időpontot követelünk meg.
- időkorlátos, ebben az esetben az esemény kezdete és befejezése közötti maximális időintervallumot határozzuk meg.

3. A Processing programozási nyelv

A Processing programozási nyelv Java alapú, de nem szükséges tudni programozni Java nyelven ahhoz, hogy akár bonyolultabb programokat hozzunk létre a nyelv keretein belül. Elsődleges célja a képzőművészeti alkotások létrehozása volt, akármilyen furcsán hangzik is ez egy programozó számára. Fő céljának tekintette és tekinti ma is, hogy a programozásban sekély tudással rendelkezők számára biztosítsa az algoritmikus megoldással létrehozható vizuális alkotások elkészítését. Vonzerőként jelentkezik, hogy segítségével megtanulható a kódolás a képzőművészet kontextusában. 2001 óta a Processing előmozdította a szoftveres írástudást a képzőművészetben belül, és a vizuális írástudást a technológián belül. Tízezrek, pl. diákok, művészek, tervezők, kutatók és egyéb érdeklődők körében használják a Processing-et tanuláshoz és alkalmazások készítéséhez. Főbb előnyei:

- Ingyenesen letölthető és nyílt forráskódú
- Interaktív programokat írhatunk benne 2D, 3D, PDF vagy SVG kimenettel
- Integrált OpenGL 2D és 3D-hez
- Elérhető GNU/Linux, Mac OS X, Windows, Android, ARM rendszereken
- Több mint 100 könyvtár bővíti az alapszoftvert
- Jól dokumentált, sok könyv elérhető

A fenti szöveg a processing.org oldalról származik. A weboldal központi szerepet játszik a

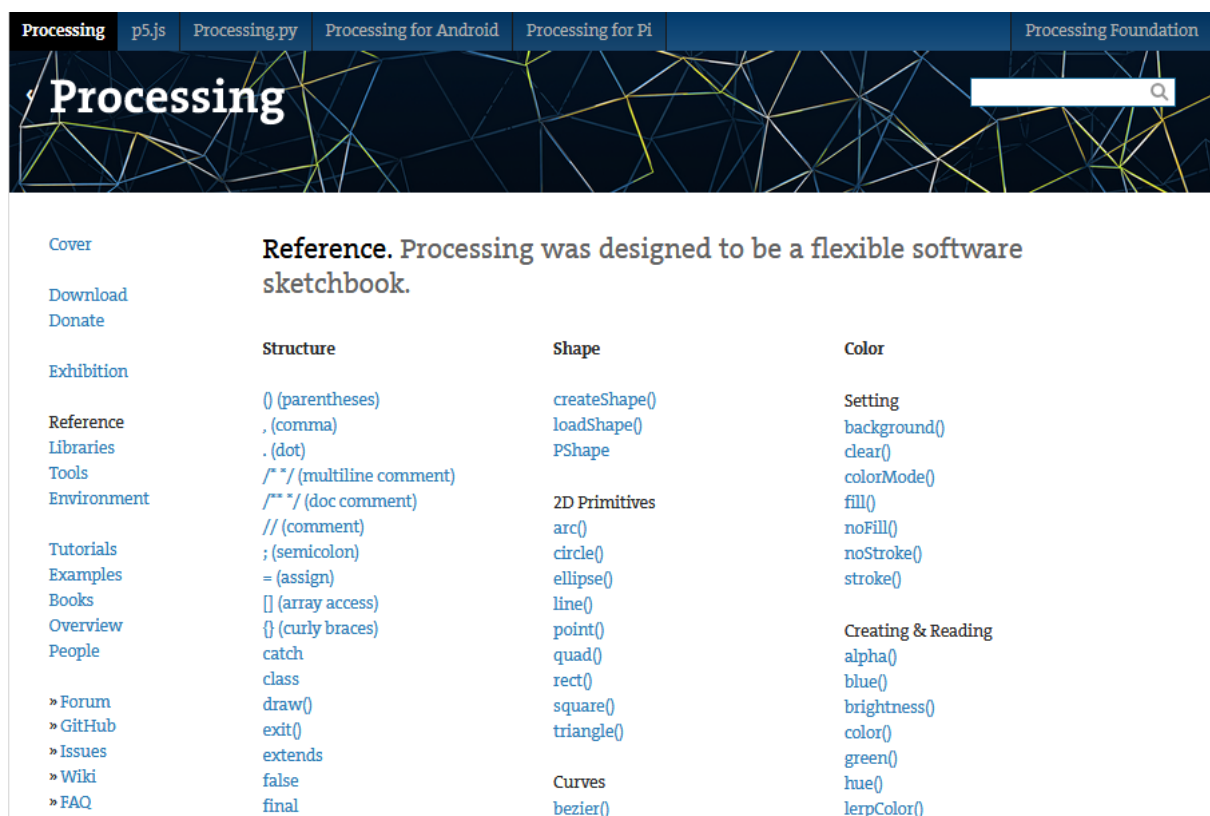


1. ábra. A Processing weboldala

Processing nyelv elsajátításában, megtalálható rajta többek között

- integrált fejlesztői környezet
- referencia oldal
- példaprogramok
- tutorialok
- könyvek

A jegyzetben szereplő összes függvény, kulcsszó, azaz a forrásprogram elemei megtalálhatók a fentebbi oldal bal oldalán fellelhető <https://www.processing.org/reference/> menüpontjában, emiatt ezek forrását külön nem fogjuk jelölni.



2. ábra. A referencia kezdőoldala

A példaprogramok önálló fejlesztéseink, szabadon felhasználhatók és módosíthatók. Ez utóbira külön biztatom az olvasót, a kísérletezés és továbbfejlesztés az egyik legjobb út egy adott nyelv, avagy a kódolás elsajátításához.

3.1. A Processing programozási nyelv alapjai

Ahogy fentebb említésre került, a Processing nyelv a Java nyelvre épül, szabadon használható minden, ami a Java-ban megszokott. Külön nem térünk ki rá, de a Processing osztályai is használhatók a Java-ban, akit érdekel ez az út, elindulhat például a <https://processing.org/tutorials/eclipse/> oldal útmutatásai alapján.



3. ábra. Processing használata Java nyelven

Ebben az esetben tetszőleges Java fejlesztésre alkalmas fejlesztői környezetben dolgozhat. Általában egy programozási nyelv gyakorlatorientált ismertetésénél a "Hello world" programot szokás megadni, de ettől most eltérünk, ugyanis a legrövidebb program már képes egy szürke hátterű ablakot megjeleníteni. Ettől rövidebb program nem létezik, mivel egy üres, tehát nulla darab karaktert tartalmazó program képes ezt a feladatot elvégezni. A forráskódot emiatt mellőzzük ebben az esetben. Mielőtt megírnánk első programunkat, a Java nyelv sajátosságait ismertetjük röviden.

3.1.1. A nyelv alapelemei

A program alkotóelemei a következők:

- Az azonosítókban betűk, számjegyek, az „_” jel és a valuta szimbólumok szerepelhetnek. Számjeggyel nem kezdődhet. Akár ékezetes betűt is tartalmazhat (unikód), de ezt inkább kerüljük.
- Kulcsszó
- Literál: állandó, amely beépül a program kódjába. Lehet:
 - egész (automatikusan int típusú, lehet hexadecimális, decimális, oktális),
 - valós (pl.: 2.3 2. .3 6e23),
 - logikai (true, false),
 - karakter (pl. '?', '\u1ac5'),
 - szöveg (pl. "valami"),
 - null,
 - Vezérlő szekvenciák: \n \t \\" \' \\",
pl. println("Egy idézet: \"Talpra magyar!\")

3.1.2. Típusok

Két fő csoportot különböztetünk meg.

1. Primitív típusok (numerikus vagy logikai)

- byte 1 byte -128 .. 127
- short 2 byte -32768 .. 32767
- int 4 byte kb -2*109 .. 2*109
- long 8 byte kb. -1019 .. 1019
- float 4 byte kb. 1.4E-45 ... 3.4E+38
- double 8 byte kb. 5E-324 ... 1.8E+308
- char 2 byte (A char is numerikus típus)
- boolean (Az egyetlen logikai típus)

2. Referencia típus: olyan mutató, mely egy objektum hivatkozását tartalmazza.

- color 4 byte

Tekintve, hogy a char is numerikus típus, teljesen szabályos a következő programrészlet:

```
char c;  
int a;  
c=65;  
println(c);  
c='A'+1;  
println(c);  
a=c+1;  
println(a);
```

A program futásának eredménye A B 67, mint a 4. ábrán is látható. A következő fejezetben ismertetjük a forrásprogramok kipróbálását. Láthatjuk, hogy a változók deklarálása deklaráló utasítással történik, tehát megadjuk a típusát és az azonosítóját. Inicializált változó deklarálás történik, ha egyben kezdőértéket is adunk, mint pl y esetében:

```
int a, b;  
double x, y=2.4;
```

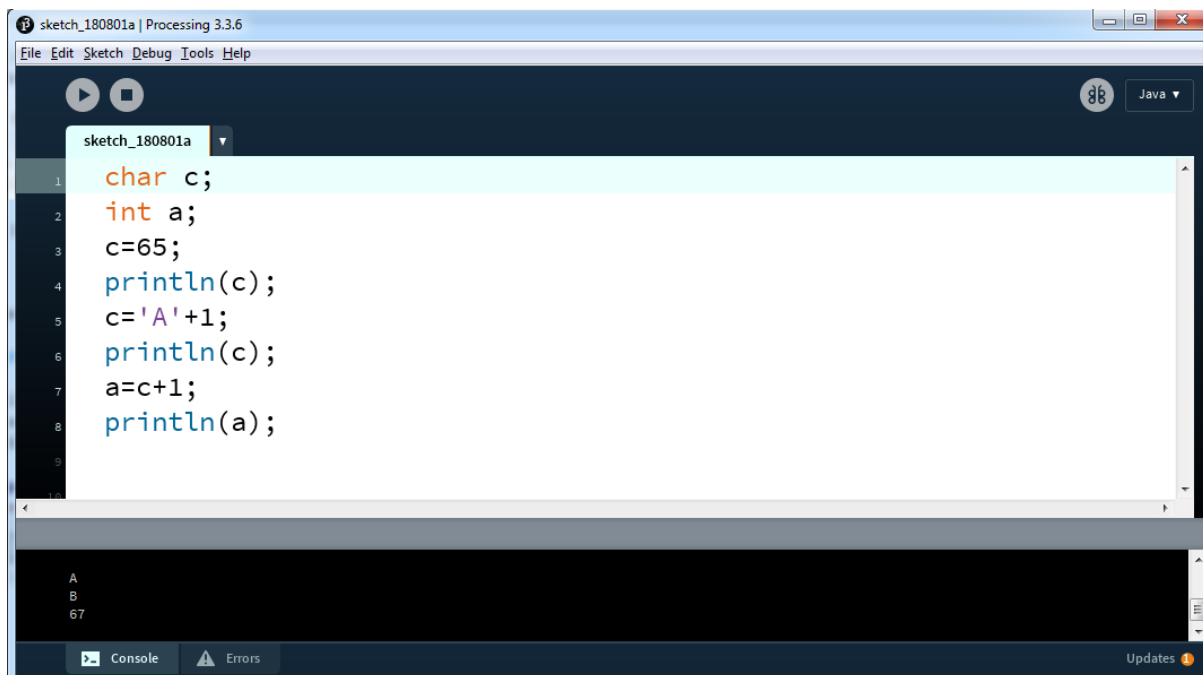
Ebben a példában látható, hogy változó deklarálásnál több változót is fel lehet sorolni, illetve a második sorban történik az ún. változó deklarálása inicializálással.

Mivel a Processing erősen típusos nyelv, fontos megismernedni a típuskonverziókkal. A típuskonverzió lehet

- automatikus (implicit)
- kényszerített (explicit): (típus) kifejezés

illetve az eredményül kapott típus alapján lehet

- szűkítő
- bővítő



4. ábra. A program és a futtatás eredménye

Alapszabály, hogy primitív típusok esetén egy szűkebb adattípus értéke konvertálható egy bővebb adattípus értékébe információ-vesztés nélkül. Ez általában automatikus. Például:

```

int i;
double d;
byte b;
short s;
println((b + s)); //implicit bővítő //konverzió (int)
d = i; //implicit bővítő konverzió
i = d; //szintaktikai hiba
i = (int)d; //explicit szűkítő konverzió

```

Numerikus operátorok típuskonverziói esetében az automatikus konverziók a következő szabályok szerint zajlanak:

- Unáris operátorok: ha az operandus int-nél szűkebb, akkor int-re konvertál, egyébként nem konvertál
- Bináris operátorok: mindkét operandust a kettő közül a bővebbre, de minimum int-re konvertálja

Az egészlitérál automatikusan int, a valós literál automatikusan double.

Az értékadó utasítással már foglalkoztunk, általános alakja a

változó = kifejezés;

A kifejezés típusának értékadás szerint kompatibilisnek kell lenni a változó típusával. Ezek a következő esetek lehetnek:

- azonos típusok, nem történik semmi

- ha a jobb oldal szűkebb, akkor implicit bővítő konverzió fog lezajlani
- ha a bal oldal byte, short vagy char, a jobb oldal int, és a fordító el tudja dönteni, hogy a jobb oldal befér a bal oldalba, akkor implicit szűkítő konverzió történik, például:

```
byte b = 100;    //az egész literál automatikusan
                 //int típusú
```

- minden más esetben fordítási hiba keletkezik

3.1.3. Operátorok

Tekintsük át a nyelv operátorait:

- Unáris (postfix/prefix) operátorok:

[] tömbképző

. minősítő

() metódus képző

~, ! bitenkénti, ill. logikai tagadás

new példányosító

(típus)kifejezés típuskényszerítő

! negáló

+, - előjel

++, – növelő, csökkentő.

Például az i++ vagy ++i közötti különbség: mindkettő növeli i értékét, de az első kifejezés értéke i eredeti, míg a második i megnövelt értéke lesz.

- Bináris operátorok

*, / szorzás, osztás

% maradékos osztás, mely valós számokra is alkalmazható. Ha az operandusok egészek, akkor az eredmény is egész, ha legalább az egyik operandus valós, akkor az eredmény is valós.

```
int x = 3;
```

```
double y = x / 2; //y értéke 1!!
```

```
y = y/2; //y értéke 0.5
```

+, - összeadás, kivonás

<, <=, >, >=, ==, != relációs operátorok, eredményük boolean típus lesz (kisebb, kisebb egyenlő, nagyobb, nagyobb egyenlő, egyenlő, nem egyenlő)

instanceof az adott osztályból való származást dönti el

&, && logikai és (teljes, ill. rövid kiértékelésű). A rövid kiértékelés csak addig vizsgál, míg el nem dől a kifejezés értéke.

!, || logikai vagy (teljes, ill. rövid kiértékelésű).

^ logikai kizáró vagy

&, |, ^ bitenkénti operátorok

<<,>>,>>> léptetések (>>>: minden esetben 0 lép be)

= értékadó operátor (lehetséges a többszörös értékadás, pl: x=y=5;)

+=, -=, *=, /=, %=, ... összetett értékadó operátorok

Az összetett és a hagyományos értékadó utasítások nem minden esetben teljesen egyformák, mert az összetett értékadásnál a változó típusát rákényszerítjük a jobb-
oldalra, tehát pl.

```
int i=0;
```

```
i=i+0.5;//hibás!!
```

```
i+=0.5; //helyes
```

- Ternáris operátorok

(logikai kifejezés) ? kifejezés1: kifejezés2 feltételes operátor

```
boolean öreg = (kor>100)?true:false;
```

Egy kifejezés kiértékelési sorrendjét meghatározza (ebben a sorrendben):

1. zárójel
2. prioritás
3. asszociativitás (balról – jobbra vagy jobbról – balra szabály)

Az alábbiakban az operátorok prioritását adjuk meg, mellettük jelölve az asszociativitás irányát.

Operátorok típusa	Prioritás (lefelé csökken)	Irány
unáris postfix	kif++ kif- [] . ()	bj
unáris prefix	++kif -kif +kif -kif !	jb
példányosítás, típus kényszerítés	new (típus)kifejezés	bj
multiplikatív	* / %	bj
additív	+ -	bj
bitenkénti eltolás	<< >> >>>	bj
relációs	< > <= >= instanceof	bj
egyenlőség	== !=	bj
logikai/bitenkénti AND	&	bj
logikai/bitenkénti XOR	^	bj
logikai/bitenkénti OR		bj
logikai rövid AND	&&	bj
logikai rövid OR		bj
ternáris	? :	bj
értékadó	= += -= *= /= %= &= ^= = <<= >>= >>>=	jb

3.1.4. Utasítások

A következő utasításokat különböztetjük meg:

- deklaráló
int a;

- értékadó
a = b * 2;
- postfix és prefix növelő és csökkentő
a++; ++a; a--; -a;
- metódushívás
println("Valamit kiirunk");
- példányosítás
auto = new Auto("ABC123");
- Programvezérlő
if, switch, while, stb...
- Üres
;
- Blokk (vagy összetett utasítás)
{ ... }

Minden utasítást pontosvessző zár le. A deklaráló, értékadó, valamint a postfix és prefix növelő és csökkentő utasításokkal az előbbiekben már foglalkoztunk. A metódusokkal és a példányosítással későbbiekben ismerkedünk meg, ezért csak a programvezérlő utasításokkal fogunk bővebben foglalkozni. Az üres utasítást ritkán használjuk, általában csak a programfejlesztés közben fordul elő. A blokkutasítás ott kerül felhasználásra, ahol a szintaktika csak egy utasítást engedélyez, ilyenkor több utasítást foghatunk össze egyé a segítségével. A programvezérlő utasítások a következők:

```
if(feltétel)
    utasítás1;
else
    utasítás2;
```

- feltétel: logikai kifejezés
- az else ág elhagyható
- a feltétel után nincs pontosvessző
- az utasítás esetén viszont van pontosvessző
- a feltétel zárójelben van, kötelező a használata
- egy ágban több utasítás is lehetséges, ekkor használjuk a fentebb említett blokkutasítást
{ ... }
- lehetséges az egymásba ágyazás

Példa:

```
if (a > b)
    c = a;
else
    c = b;
```

Azaz ha a értéke nagyobb, mint a b, akkor c legyen egyenlő a-val, különben pedig c legyen egyenlő b-vel.

```
switch(kifejezés){
    case érték1: utasítások;
    case érték2: utasítások;
    ...
    default:    utasítások;
}
```

- akkor alkalmazható, ha egy kifejezés jól meghatározott, különálló értékeire szeretnénk bizonyos utasításokat végrehajtani
- kifejezés: byte, short, int vagy char, String
- az utasítások között előforduló break utasítás hatására a switch blokk végére kerül a vezérlés, e nélkül a következő case ágra kerül a vezérlés
- egy case kulcsszóhoz csak egy érték tartozhat

Általános alakja:

```
switch(kifejezés){
    case érték1: utasítások;
        break;
    case érték2: utasítások;
        break;
    ...
    default:    utasítások;
}
```

Írjunk program részletet, mely pont alapján jegyet ad a dolgozatra.

```
switch(pont){
    case 1:
    case 2:
    case 3: println("Elegtelen");
    println("Keszuljon tovabb!");
    break;
    case 4: System.out.println("Elegseges");
    break;
    case 5: System.out.println("Kozepes");
    break;
    case 6: System.out.println("Jo");
    break;
    default:    System.out.println("Jeles");
}
```

A következőekben a ciklus utasítások kerülnek ismertetésre.

```
while(feltétel)
    utasítás;
```

- Amíg a feltétel igaz, újból végrehajtja az utasítást, ha hamis, akkor a ciklust követő utasításra lép.
- Több utasítás esetén: blokk { ... }

Például bankba tesszük a pénzünket kamatozni, és addig tartjuk bent, amíg milliomosok nem leszünk. A programrészlet azt számolja ki, hogy hány évet kell várnunk, ha a kamat 2 százalék. A változók deklarációit mellőztük.

```
penz=423356;
ev=0;
while(penz<1000000){
    penz*=1.02;
    ev++;
}
println(ev+" ev múlva leszunk gazdagok");
```

```
do
    utasítás;
while(feltétel);
```

- Amíg a feltétel igaz, újból végrehajtja az utasítást, ha hamis, akkor a ciklust követő utasításra lép.
- Több utasítás esetén: blokk { ... }

A feladat ugyanaz...

```
penz=423356;
ev=0;
do{
    penz*=1.02;
    ev++;
}while(penz<1000000);
println(ev+" ev múlva leszunk gazdagok");
```

Fontos különbség, hogy ez a ciklus egyszer mindig le fog futni, emiatt ha a kezdőtőke eleve nagyobb, akkor is egy évet ír ki, tehát csak megszorításokkal működik helyesen a program.

```
for(inicializálás; feltétel; léptetés)
    utasítás;
```

- inicializálás: egy vagy több utasítás vesszővel elválasztva, mely egyszer hajtódik végre a ciklusmagba való első belépés előtt. Pl. ciklusváltozó deklarálása, inicializálása. A ciklusváltozó típusa tetszőleges.
- feltétel: logikai kifejezés. Amíg igaz, újból végrehajtja az utasítást, ha hamis, akkor a ciklust követő utasításra lép.

- léptetés: egy vagy több utasítás vesszővel elválasztva, mely(ek) a ciklusmag minden egyes lefutása után automatikusan végrehajtódik. Általában a ciklusváltozót szokás itt növelni, vagy csökkenteni.
- A while ciklus egy speciális esetének tekinthető, a

```
for(inicializálás; feltétel; léptetés)
    utasítás;
```

hatása megegyezik az alábbi részlet hatásával:

```
inicializálás;
while(feltétel){
    utasítás;
    léptetés;
}
```

Írass ki 10 db csillagot!

```
for(int i=1; i<=10; i++)
    print(" *");
```

Írass ki 10 sorba csillagokat, mindegyik sorban 10 db legyen!

```
for(int i=1; i<=10; i++){
    for(int j=1; j<=10; j++)
        print(" *");
    println();
}
```

Írass ki a nagybetűs angol ábécét!

```
for(char c='A'; c<='Z'; c++)
    print(c+" ");
```

(c+' ') esetén a karakterek kódjait írja ki!

Írass ki a nagy- és kisbetűket az angol ábécéből!

```
for(char n='A', k='a'; n<='Z'; n++, k++)
    print(n+" "+k+" ");
```

Beteszünk a bankba 500 000 Ft-ot éves 10% kamatra. Írass ki az évenkénti kamatozott összeget, amíg nem érjük el az 1 000 000 Ft-ot!

```
for(double d=500000; d<=1000000; d*=1.1)
    println(d);
```

A switch esetében már említésre került a break utasítás, amivel ki lehet lépni a blokkból. Összegezzük ezeket a vezérlés átadó utasításokat:

- break: az aktuális utasításblokkból (pl. ciklusból) való azonnali kiugrást eredményezi.
- continue: hatására a vezérlés az utasításblokk (ciklus) végére kerül, s folytatja a következő iterációval.

3.1.5. Függvények, vagy más néven metódusok

A metódusra a nevével és aktuális paramétereivel kell hivatkozni. Lehet eljárás- vagy függvényszerű. Az eljárásyszerű metódusok csak tevékenységet végeznek, a függvényszerűek ezen felül még értéket is előállítanak.

- Eljárásyszerű metódus: visszatérési értékének típusa void, azaz üres típus, nem tér vissza értékkel. Tartalmazhat "return;" utasítást, ekkor véget ér a metódus törzsének végrehajtása, de ezt általában nem alkalmazzuk
- Függvényszerű metódus: visszatérési értékének típusa valamilyen void-tól különböző típus, azaz "igazi" értékkel tér vissza. Muszáj "return kifejezés;" utasítást tartalmaznia a metódusból való kilépés előtt. A kifejezés típusa megegyezik a metódusfejben megadott visszatérési érték típusával
- A függvényszerű metódus eljárásként is hívható

Túlterhelés (overloading): lehet több azonos nevű metódus, melyek a paraméterezésben, és esetleg a visszatérési érték típusában térhetnek el egymástól. Például:

```
float max(float a, float b)
int max(int a, int b)
void vonalhuz()
int vonalhuz() //Nem lehet csupán visszatérési
               //típusban eltérnie! Hiba!!
int vonalhuz(int hossz)
```

A metódus szintaktikája:

```
[módosítók] visszatérésitípus azonosító([paraméterlista]){
Utasítások;
}
```

Processing-ben általában nem használjuk a Java-ban megszokott módosítókat (public, private, protected, stb.).

A paraméterátadás érték szerint történik, ezért az aktuális paraméter típusának értékadás szerint kompatibilisnek kell lennie a formális paraméterrel. A visszatérés a metódusból függvényszerű metódus esetén a "return;" utasítás után történik. A return után kötelezően meg kell adnunk egy értéket eredményező kifejezést. További jellemzők:

- A sorrend tetszőleges, de van két kitüntetett metódus, a setup() és a draw().
- Nem ágyazhatók egymásba.
- Lehet rekurzív metódusokat is definiálni.
- A metódusban –mint egyébként is bármely blokkban– definiálhatunk lokális változót.

Először általában az importálandó osztályokat adjuk meg, ha vannak, ez csak később kerül előtérbe. Utána a globális, mindenhol elérendő változók megadását szoktuk elvégezni, majd következik a setup() és a draw() függvény, s utána a többi függvény.

Tehát alapesetben a Processing forrásprogram két függvényt fog tartalmazni: setup() és draw().

3.1.6. A setup() függvény

A referencia oldalon minden függvény, kulcsszó, stb. esetében fel van tüntetve a leírás, az esetleges paraméterek jelentése és típusai, valamint egy rövid, könnyen érthető forrásprogram, vagy annak egy jellemző részlete. Ezek a példák közvetlenül beemelhetők a fejlesztői környezetbe, kipróbálhatók.

Exhibition	Name	setup()
Reference Libraries Tools Environment Tutorials Examples Books Overview People	Examples	<pre>int x = 0; void setup() { size(200, 200); background(0); noStroke(); fill(102); } void draw() { rect(x, 10, 2, 80); x = x + 1; }</pre>
» Forum » GitHub » Issues » Wiki » FAQ » Twitter » Facebook » Medium		<pre>int x = 0; void setup() { fullScreen(); background(0); noStroke(); fill(102); } void draw() { rect(x, height*0.2, 1, height*0.6); x = x + 2; }</pre>
	Description	The <code>setup()</code> function is run once, when the program starts. It's used to define initial environment properties such as screen size and to load media such as images and fonts as the program starts. There can only be one <code>setup()</code> function for each program and it shouldn't be called again after its initial execution.

5. ábra. A setup() függvény leírása a Processing oldalán

A setup() függvény automatikusan, és pontosan egyszer fut le, amikor a program elindul, nem szabad tehát meghívni. Arra szolgál, hogy meghatározza a kezdeti környezeti tulajdonságokat, például a képernyő méretét, vagy képeket és/vagy betűkészleteket töltsön be a program indulásakor. Minden programban csak egy setup() függvény lehet.

Ha az alkalmazás ablakmérete eltér az alapértelmezett mérettől, akkor a setup első sorának a size() vagy a fullScreen() függvénynek kell lennie. Többnyire ez így szokott lenni, mert az alapértelmezett méret operációs rendszer függő, de minimum 100x100-as, ettől kisebbet nem lehet beállítani sem.

Megjegyzés: A setup()-ban deklarált változók sem érhetők el más függvényekből, emiatt ne itt deklaráljuk őket, később látunk majd sok-sok példát erre.

Példa a setup() függvényre:

```
void setup() {
  size(800, 600);
  background(100);
}
```

A size() az ablakban lévő rajzterület méretét állítja be, meghatározza a szélességét és magasságát pixelben, csak egyszer hívható és nem használható átméretezéshez. A függvény a width

és a height ún. beépített változók értékeit állítja be. Értékeiket tudjuk olvasni, de egyéb módon nem tudjuk állítani. Ha nincs size(), akkor 100 lesz az értékük. Teljes képernyős futtatáshoz a fullscreen() használható. Ha a beállított érték nagyobb, mint a minimális ablakméret, akkor az ablakméret idomul a rajzterület méretéhez.

A background() az ablak háttérszínét állítja, sokszor használjuk ablak törlésre is a draw()-ban. Az alapértelmezett háttér világosszürke. A referenciát böngészve látható, hogy nem csak egy paraméteres változata van, azaz túl van terhelve a függvény, ez Java-ból jól ismert lehetőség:

- background(rgb)
- background(rgb, alpha)
- background(gray)
- background(gray, alpha)
- background(v1, v2, v3)
- background(v1, v2, v3, alpha)
- background(image)

A teljességre nem törekedve, egy darab egész paraméter esetén szürke árnyalatot kapunk, három esetén RGB kóddal van dolgunk. Mindegyik esetben 0 és 255 közötti értékeket használunk.

3.1.7. A draw() függvény

Közvetlenül a setup() után fut le. A draw() újra és újra végrehajtja a blokkjában található kódot, amíg a program le nem áll vagy a noLoop() meg nem hívódik. Hívása automatikusan történik, tilos meghívni. Minden program csak a draw() végén frissíti a képernyőt, hamarabb nem változik meg az ablak tartalma.

A frameRate() függvénnyel befolyásolhatjuk a másodpercenkénti rajzolások számát, azaz a draw() meghívási gyakoriságát (alapértelmezetten 30-szor fut le). Ha folyamatosan futó programot írunk, akkor lenni kell draw()-nak a forrásban, de lehet üres is, ha pl. eseményvezérelten kezeljük a kirajzolásokat.

Példa draw() függvényre:

```
void draw(){
    fill(100, 0, 0);
    rect(100, 100, 200, 150);
}
```

A fill() függvény az alakzatok kitöltéséhez használt színt állítja be. Például, ha meghívjuk a 204, 102, 0 paraméterekkel, akkor minden további alak narancssárgával lesz kitöltve. A szín függ a colorMode()-dal állítható módtól, lehet RGB vagy HSB. Az alapértelmezett színmód RGB, minden érték 0 és 255 között van.

Ha egy szín megadásához hexadecimális jelölést használ, használja a "#" vagy a "0x" karaktereket az értékek előtt (például #CCFFAA vagy 0xFFCCFFAA). A # szintaxis hat számjegyet használ a szín meghatározására. A "0x" karakterekkel kezdődő hexadecimális jelölés használatkor a hexadecimális értéket nyolc karakterrel kell meghatározni; az első két karakter az alfa, a többi pedig a piros, a zöld és a kék összetevőt határozza meg. A fill() is túl van terhelve:

- fill(rgb)
- fill(rgb, alpha)
- fill(gray)
- fill(gray, alpha)
- fill(v1, v2, v3)
- fill(v1, v2, v3, alpha)

A rect() függvény téglalapot rajzol a képernyőre. Alapértelmezés szerint az első két paraméter a bal felső sarok helyét, a harmadik a szélességet, a negyedik a magasságot állítja be. A paraméterek értelmezésének módja azonban megváltoztatható a rectMode() függvénnyel. Átláthatósági szempontok miatt előre a setup() szokott kerülni, s ezt követi a draw(). Tehát a következő program egy darab piros téglalapot fog kirajzolni egy szürke hátterű ablakba, s mindaddig futni fog, míg manuálisan meg nem szakítjuk a végrehajtását:

```
void setup(){
    size(800,600);
    background(100);
}

void draw(){
    fill(100, 0, 0);
    rect(100, 100, 200, 150);
}
```

3.2. Az IDE

Az integrált fejlesztői környezet (IDE) letölthető a weboldal Download menüpontjából. Kiválasztva a megfelelő operációs rendszert, komplett környezetet tudunk telepíteni, melyben lehetőségünk nyílik szerkeszteni, fordítani, futtatni. Az IDE a modernebb rendszerekhez képest szegényesebbnek tűnik, de a célnak megfelel.

Az újabb verziók általában félévente jelennek meg, erről a weboldalon pontos információk állnak rendelkezésre. A program telepítése a szokásos, a fejlesztői környezettel együtt felkerül a fordításhoz szükséges környezet is, külön ezzel nem kell foglalkozni. A program elindítása után egy minimalista jellegű szerkesztői felülettel találkozunk. A szokásos menüpontokon túl érdemes megjegyezni az Edit/Auto Format lehetőséget, ez nagyban megkönnyíti időnként a munkánkat. Lehetőségünk van "crossplatformos" munkára, egyrészt a File/Export Application teszi lehetővé, másrészt pedig az ablak jobb felső részén elhelyezkedő Java feliratú legördülő menü, ahol az Android mellé egyéb lehetőségeket tudunk választani az Add Mode használatával. Itt most ezekkel a továbbiakban nem foglalkozunk. A programjaink gyors indítását és megállítását egyszerűen elérhetjük, nem kell lenyitni a legördülő menüket. Fentebbiek alapján az üres szerkesztői ablak ellenére már kipróbálhatjuk a rendszert. Elindítva az egyetlen karaktert sem tartalmazó programunkat, megjelenik egy minimum 100x100-as méretű szürke hátterű ablak, melyet vagy a stop gombbal, vagy pedig az ablak lezáró szimbólumával tudunk bezárni. Próbáljuk ki, hogy meggyőződünk a telepítés működőképességéről. Amíg nem működik, felesleges bonyolultabb kódot írni.



6. ábra. A Processing letöltési felülete



7. ábra. A programok gyors kezelését lehető tévő gombok

Az ablak alsó részén fognak megjelenni a hibaüzenetek, valamint az ún. konzolos kiírásaink is. Írjuk be az alábbi sorokat, s futtassuk le a programot. Gépelés közben figyeljük az említett területet, s láthatjuk, ahogy folyamatosan folyik a szintaktikai ellenőrzés és megjelennek a hibaüzenetek.

```
background(0,255,0);
print("Hello vilag! Rajz merete: "+width+"x"+height);
```

Figyeljük meg a 8. ábrán, hogy az ablak minimális mérete nagyobb, mint 100x100 a futtatási környezet miatt.

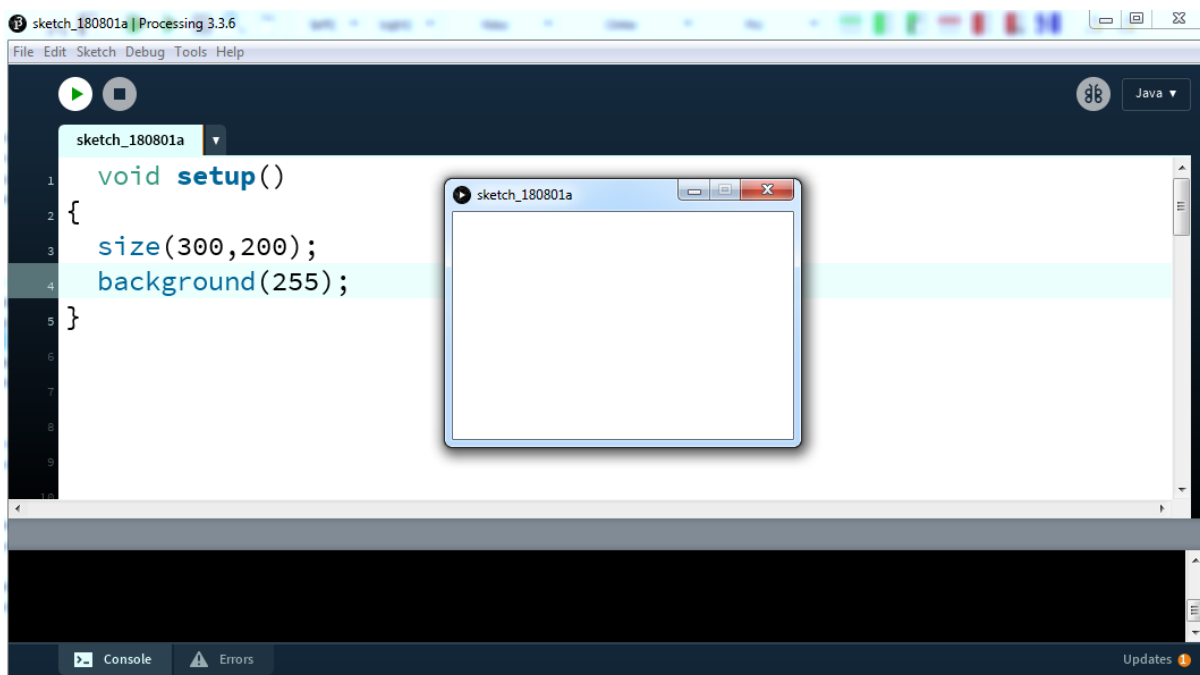
3.3. Egy egyszerű feladat megoldása

Lépésenként fogjuk megoldani a következő feladatot:

Egy 300x200-as fehér háttérű ablakban mozgassunk egy kék 50x40-es téglalapot a bal szélétől a jobb széléig mozgás közben -függőlegesen- középre pozicionálva. A programozást lépésekre bontjuk. Először csak az ablakot rajzoltatjuk ki helyes háttérrel. A következő a téglalap megjelenítése a bal szélén. Gondot jelent, hogy a setup() csak egyszer fut le, emiatt nem mozoghat a téglalap. Próbáljuk meg ciklussal mozgatni.

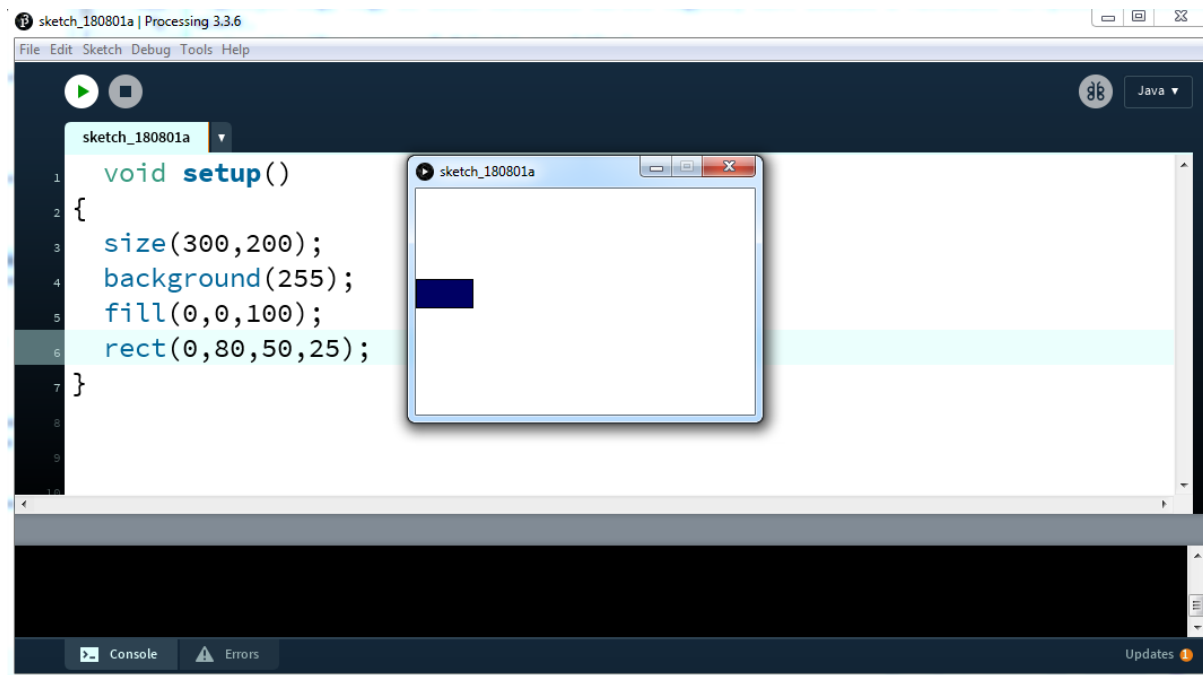


8. ábra. A Processing fejlesztői környezete használat közben



9. ábra. Az ablak méret- és színhelyesen

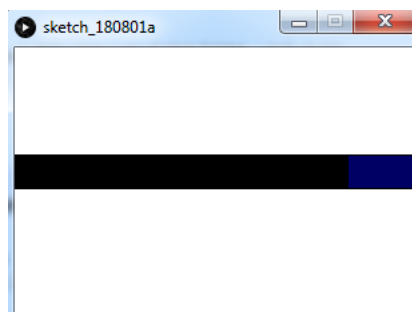
```
void setup() {
  int x;
  size(300, 200);
  background(255);
  fill(0, 0, 100);
  for (x=0; x<250; x++) {
    rect(x, 80, 50, 25);
  }
}
```



10. ábra. Az ablak a téglalappal

```
}
}
```

Az eredmény a következő ábrán látható, egyrészt nem látjuk a mozgást, másrészt rossz a kimenet.



11. ábra. Az ablak a mozgó téglalappal

Próbálkozhatunk pl. üres ciklussal való lassítással, mozgás utáni törléssel:

```
void setup() {
  int x;
  size(300, 200);
  background(255);
  fill(0, 0, 100);
  for (x=0; x<150; x++) {
    background(255);
    for (long y=0; y<1000000; y++);
    rect(x, 80, 50, 25);
  }
}
```

```
}
```

Idővel belátjuk, hogy a `setup()` nem erre való, használjuk a `draw()` metódust, megspóroljuk vele a ciklust, átláthatóbb a program, leegyszerűsödik a munkánk:

```
int x=0;
void setup(){
  size(300, 200);
  fill(0, 0, 100);
}
void draw(){
  background(255);
  rect(x++, 80, 50, 25);
  if (x>250) noLoop();
}
```

Az `x` változó deklarációja a függvényeken kívül van. Ha `setup()`-on belül lenne, a `draw()` nem éri el, ha a `draw()`-ban helyezzük el, akkor lenullázódott volna minden egyes esetben, amikor a `draw()` újra lefut. A `size()` és a `fill()` maradt a `setup()`-ban, mert csak egyszer kell hívni őket, viszont a `background()` átkerült, ezzel biztosítjuk, hogy a régi helyzetű téglalap törlésre kerüljön, ne egymásra rajzolja őket. A program leállítását a `noLoop()` biztosítja.

4. Egyszerű időfüggő feladatok kódolása Processing környezetben

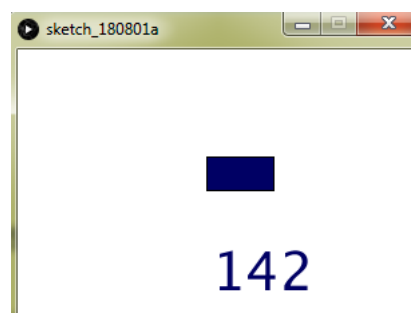
Az iménti feladatban az animáció sebességét többféleképpen is befolyásolhatjuk, pl. `x` kisebb/nagyobb arányú növelésével, vagy időhöz kötéssel, vagy a másodpercenként lejátszandó képkockák számával, amit a `frameRate()` függvénnyel tudunk kezelni. A továbbiakban mi a valós időhöz kötjük a programjainkat, azaz a számítógép belső idejéhez.

4.1. Digitális óra

A feladunk egy digitális óra elkészítése. Grafikus felületen szeretnénk kiíratni, nem a konzolra. Ehhez a `text()` függvényt használjuk fel. Első paramétere a kiírandó szám vagy szöveg, a második és harmadik paramétere pedig a kiírás helyét adja meg. A színt a már megismert `fill()` határozza meg. A karakterek nagyságát a `textSize()` függvénnyel tudjuk befolyásolni. Pl. kiegészítve az előbbi programkódot, írassuk ki az `x` változó értékét az ablakba!

```
int x=0;
void setup(){
  size(300, 200);
  fill(0, 0, 100);
  textSize(40);
}
void draw(){
  background(255);
  rect(x++, 80, 50, 25);
  text(x, 145, 180);
  if (x>250) noLoop();
}
```

A program futása során az alábbiak kell megjelennie. A kiíratáshoz tudnunk kell a pontos



12. ábra. A pozíció kiíratása

időt. Az Processing időlekérdező függvényei "növekvő" sorrendben:

- `millis()`: a program indítása óta eltelt idő ezredmásodpercben
- `second()`: másodperc
- `minute()`: perc
- `hour()`: óra

- day():nap
- month():hónap
- year():év

Ezek után már egyszerű megírni a programot:

```
void setup()
{
  size(300, 400);
  textSize(32);
  fill(0);
}

void draw()
{
  background(255, 255, 0);
  text(nf(hour(), 2) + " " + nf(minute(), 2) + " " + nf(second(), 2),
    100, 200);
}
```

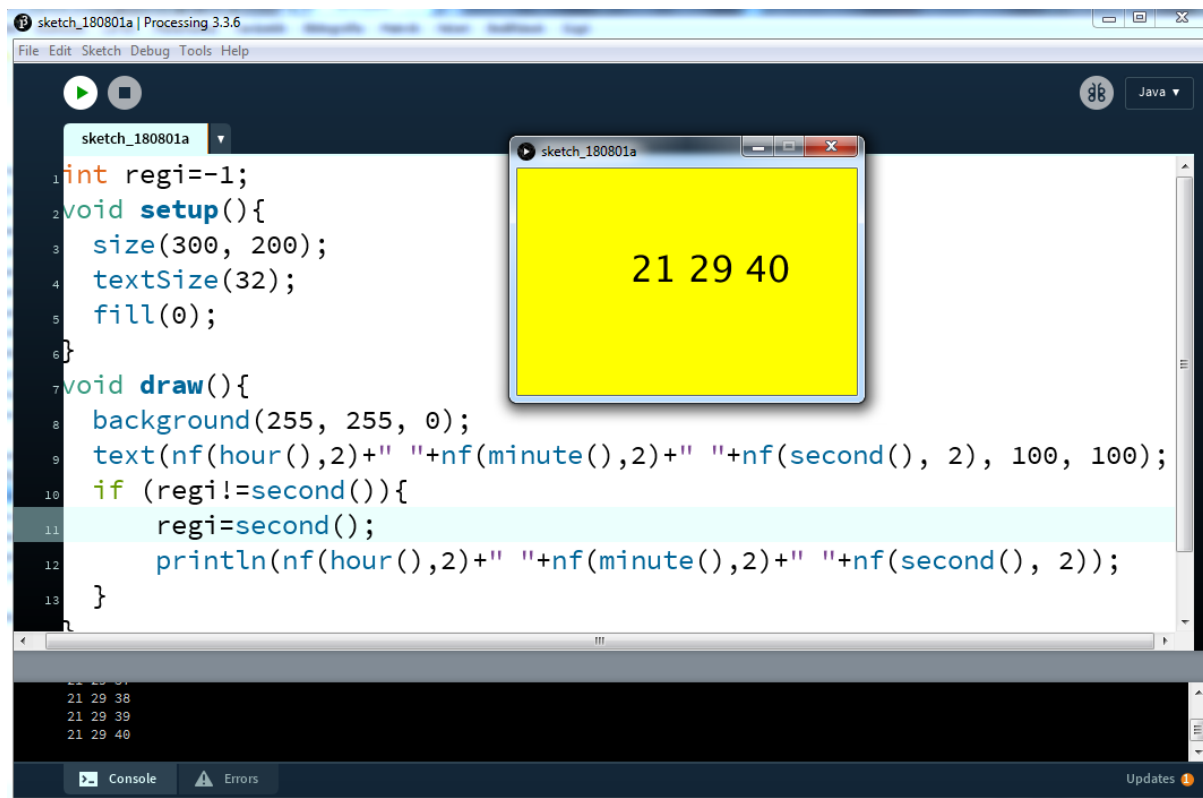
Az nf() függvényt azért érdemes használni, hogy szebb legyen az eredmény, segítségével 5 helyett 05 íródik ki. Érdekesebb az, hogy miként tudnánk kiírni a konzolra az időt. A problémát az jelenti, hogy míg grafikus képernyőn nem feltűnő, hogy másodpercenként többször is kiíródik ugyanaz a szám, konzolon ez már nem igaz. Konzolra csak akkor írássunk, ha változott az idő! Egy változóban tartjuk nyilván a már kiírt másodpercet, kezdő értéke -1, így már az első draw() esetén ír konzolra is a program. A megoldás a 13. ábrán látható.

4.2. Analóg óra

Adódik a következő feladat, szeretnénk hagyományos formában látni az időt. Ehhez tudnunk kell kört és szakaszt rajzolni a képernyőre. A kör rajzolásához az ellipse()-t fogjuk használni. Négy egész paramétere van, az első kettő a középpont koordinátáját, a második kettő pedig az ellipszis szélességét és magasságát adja meg, tehát a tengelyei párhuzamosak a koordináta tengelyekkel. Kitöltésére a fill()-t, körvonala színének meghatározására a stroke() függvényt használjuk. A stroke() a szakaszok színét is állítja. A körvonalak/szakaszok vastagságát a strokeWeight() függvénnyel állíthatjuk be, alapértelmezett az egy pixeles vastagság. A körhöz használhatnánk a circle() függvényt is, de ha szeretnénk ellipszis alakú órát, akkor érdemes egyből az ellipse()-ben gondolkozni, hogy később ne kelljen átírni a programot. A circle() függvény hasonló paraméterezésű, az első két paraméter a kör középpontja, a harmadik pedig az átmérője. Kitöltését, körvonal színét és vastagságát szintén a fill() és a stroke(), strokeWeight() befolyásolja.

A számlap elkészítéséhez szükségünk van egy körre, ami nem okozhat gondot, valamint a számok felírására. A számok pozíciójának meghatározásához ismerni kell a középpont koordinátáját, valamint a központi szöget és a sugarat. Ezek segítségével és pl. a szinusz, koszinusz függvények felhasználásával már megoldható a probléma. A Processing nyelvben egyszerűen használhatók ezek a függvények, leírásukat a Trigonometry részben találhatjuk meg a referencia oldalon. Tehát adottak a következő implementációk:

- acos(): koszinusz inverze



13. ábra. A pontos idő kiírása grafikusán és konzolra

- `asin()`: szinusz inverze
- `atan()`: tangens inverze
- `atan2()`: adott pontból az origóhoz húzott szakasz szögét adja vissza
- `cos()`: koszinusz
- `degrees()`: radiánból fokba alakít
- `radians()`: fokból radiánba alakít
- `sin()`: szinusz
- `tan()`: tangens

A szögfüggvények alapértelmezetten radiánban számolnak. Más matematika eszközeink is vannak, lsd. a Calculation részt a referenciában (a gyakrabban használtakat soroljuk csak fel):

- `abs()`: Abszolút érték
- `ceil()`: Felső egészrész
- `constrain()`: Intervallumba kényszeríti egy változó értékét
- `dist()`: Két pont távolsága
- `exp()`: Exponenciális

- floor(): Alsó egészrész
- log(): Logaritmus
- max(): Maximum
- min(): Minimum
- pow(): Hatványozás
- round(): Kerekítés
- sq(): Négyzetre emelés
- sqrt(): Négyzetgyök vonás

Amennyiben további matematikai függvényekre van szükség, bátran használjuk a Java nyelv Math osztályát, mely pl. a <https://docs.oracle.com/javase/7/docs/api/> oldalon érhető el. Könnyen megtaláljuk a 14. ábrán látható Math osztály leírását, ahonnan továbblépve részletesebben megismerhetjük a metódusokat (ld. 15. ábra).

Field Summary	
Fields	
Modifier and Type	Field and Description
static double	<code>E</code> The double value that is closer than any other to <i>e</i> , the base of the natural logarithms.
static double	<code>PI</code> The double value that is closer than any other to <i>pi</i> , the ratio of the circumference of a circle to its diameter.

Method Summary	
Methods	
Modifier and Type	Method and Description
static double	<code>abs(double a)</code> Returns the absolute value of a double value.
static float	<code>abs(float a)</code> Returns the absolute value of a float value.
static int	<code>abs(int a)</code> Returns the absolute value of an int value.
static long	<code>abs(long a)</code> Returns the absolute value of a long value.
static double	<code>acos(double a)</code> Returns the arc cosine of a value; the returned angle is in the range 0.0 through <i>pi</i> .
static double	<code>asin(double a)</code>

14. ábra. A Math osztály referencia lapja

Szükséges lehet még a Π értékének használata, a referenciában a konstansoknál (Constants) találjuk meg:

- HALF_PI: $\Pi/2$
- PI: Π
- QUARTER_PI: $\Pi/4$
- TAU: 2Π
- TWO_PI: 2Π

min
<pre>public static int min(int a, int b)</pre>
Returns the smaller of two int values. That is, the result is the argument closer to the value of <code>Integer.MIN_VALUE</code>. If the arguments have the same value, the result is that same value.
Parameters: a - an argument. b - another argument.
Returns: the smaller of a and b.
min
<pre>public static long min(long a, long b)</pre>
Returns the smaller of two long values. That is, the result is the argument closer to the value of <code>Long.MIN_VALUE</code>. If the arguments have the same value, the result is that same value.
Parameters: a - an argument. b - another argument.
Returns: the smaller of a and b.

15. ábra. A min() metódus referencia lapja

π IS WRONG!

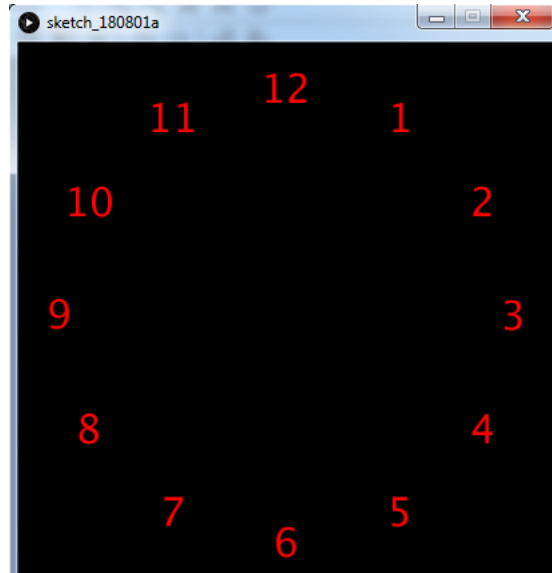
"I know it will be called blasphemy by some, but I believe that π is wrong."

-- Bob Palais

What is a circle?	A circle is defined as all points in a plane a certain distance (the radius) away from a center point. The unit circle has a radius of 1, not a diameter of 1.
Fact: The radius is more fundamental to circles than diameter. So why define the circle constant in terms of diameter?	
Better definition for circle constant	Old definition for circle constant
τ $\tau \equiv \frac{\text{circumference}}{\text{radius}}$	π $\pi \equiv \frac{\text{circumference}}{\text{diameter}}$
With τ , angles and radians are less awkward to learn and use!	
1 circle is simply 1 τ radians and 1 turn	1 circle is 2 π radians?
Euler's identity is simpler and makes geometrical sense with τ .	
Euler's identity with τ $e^{i\tau} = 1$ The complex exponential of the circle constant is unity . Geometrical: A rotation by one turn is 1.	Euler's identity with π $e^{i\pi} = -1$ The complex exponential of the circle constant is -1. Not as elegant geometrically compared to τ .
The number $\tau = 2\pi$ appears in many common formulas (note the 2).	
Gaussian distribution	Cauchy's integral formula
$\frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$	$f(a) = \frac{1}{2\pi i} \oint \frac{f(z)}{z-a} dz$
Stirling's approximation	n'th roots of unity
$n! \sim \sqrt{2\pi n} \left(\frac{n}{e}\right)^n$	$\left\{ e^{i2\pi k/n} \right\}$
And: angular frequency, reduced Planck constant, Fourier series formulas, the Gauss-Bonnet and Picard theorems, the 2n theorem, periods of sine and cosine, the sum of the exterior angles of any polygon, etc.	
The circumference and area of a circle becomes more standard with τ .	
circumference and area of a circle	velocity and displacement after constant acceleration
τr $\frac{1}{2}\tau r^2$	at $\frac{1}{2}at^2$
arc length and area of a circular sector	momentum and kinetic energy
θr $\frac{1}{2}\theta r^2$	mv $\frac{1}{2}mv^2$
base and area of a skinny triangle	spring force applied and spring potential energy
ϕr $\frac{1}{2}\phi r^2$	kx $\frac{1}{2}kx^2$
The pattern "(1/2) * constant * variable squared" is naturally occurring and shows up in various situations.	
Why are you still using π ? Let's all say "bye to π " and "embrace the τ !"	
Summary Sheet by Spiked Math (March 2012)	1. J. Lindenberg. "Tau Before It Was Cool" [sites.google.com/site/taubeforeitwascool]. 2. R. Palais. "n is Wrong!". The Mathematical Intelligencer 23 (3): 7-8, 2001. 3. M. Hartl. "The Tau Manifesto" [tauday.com]. 4. Wikipedia [en.wikipedia.org/wiki/Tau_(2pi)]

16. ábra. 2Π avagy τ

Bizonyára feltűnt, hogy a 2π értékre két konstans is rendelkezésre áll, ennek tudománytörténeti okai vannak (<http://www.math.utah.edu/palais/pi.pdf>). A 16. ábra rávilágít ennek okára. Térjünk vissza az eredeti problémára, az analóg óra megjelenítésére. A számok kirajzolásához felhasználjuk a `textAlign()` függvényt, ez biztosítja, hogy a szám középpontja egybeesik a tervezett helyével.



17. ábra. A számlap

```
void setup() {
  size(400, 400);
  background(0, 0, 0);
  int a=400;
  fill(255, 0, 0);
  textSize(30);
  textAlign(CENTER, CENTER);
  for (int i=1; i<13; i++)
    text(i, (a/2-30)*cos(radians(i*30-90))+a/2, (a/2-30)*sin(
      radians(i*30-90))+a/2);
}
```

Az eredmény a 17. ábrán látható. Ha kicseréljük az a változó értékadását úgy, hogy 400 helyett a `width` ún. beépített változót használjuk, akkor egy másik méretnél elegendő csupán a `size()` függvényben átírni a méretet. Ha nem csak négyzet alakú ablakot akarunk megengedni, akkor ugyanezt megtehetjük a másik irányban is a `height` változóval.

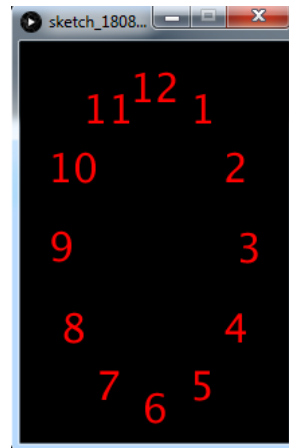
```
void setup() {
  size(200, 600);
  background(0, 0, 0);
  int a=width, b=height;
  fill(255, 0, 0);
  textSize(30);
  textAlign(CENTER, CENTER);
  for (int i=1; i<13; i++)
```

```

text(i, (a/2-30)*cos(radians(i*30-90))+a/2, (b/2-30)*sin(
radians(i*30-90))+b/2);
}

```

A változókat csak azért vezettük be, hogy ne kelljen annyit gépelni. Az óra képe így már a 18. ábrán egy ellipszis lesz. Ha a circle() függvényt használtuk volna, akkor ezt nem tudtuk volna elérni.



18. ábra. Ellipszis számlap

A továbbiak programozása innentől már könnyű lenne, emiatt annyit nehezítettünk a feladaton, hogy visszafelé járjon az óra, amihez természetesen a számlap is idomul. A mutatók megrajzolásához ki kell számolni a központi szögüket, kezdő pontjuk az ablak közepe, a végpont meghatározása pedig a számok elhelyezésénél használt matematikára alapszik, csak pl. 60-at vonunk le 30 helyett az $(a/2-30)*\cos(\text{radians}....)$ helyen, hogy körlapon belül maradjon. A helyes arányokat adott méretnél "kikísérletezhetjük", hogy **mutatós** legyes. Törekedhetünk arra, hogy tetszőleges ablakméretnél működjön, ekkor viszont a számok elhelyezésénél pl. betűméretet sem ártana állítani az ablak méretének függvényében, s nem konkrét számokkal dolgoznánk, hanem arányokkal. A fent említett esetben például nem 30-at vonnánk le, hanem valamilyen arányszámmal szoroznánk, pl. 0.85-tel: $(a/2*0.85)*\cos(\text{radians}...$ Egy lehetséges, de nem minden fenti felvetést megvalósító megoldás forráskódja lentebb látható, az eredmény pedig a 19. ábrán. A mutatók, azaz a szakaszok megrajzolásához a line() függvényt használjuk, első két paramétere a kezdőpont, második két paramétere a végpont. Figyeljük meg, hogy a perc és a másodperc mutatója ugrik, az óráé viszont finomabban mozog.

```

int a, b;
void setup() {
  size(400, 600);
  background(0, 0, 0);
  a=width;
  b=height;
  fill(255, 0, 0);
  textSize(30);
  textAlign(CENTER, CENTER);
  for (int i=1; i<13; i++)
    text(i, (a/2-15)*cos(radians(-i*30-90))+a/2, (b/2-15)*
sin(radians(-i*30-90))+b/2);
}

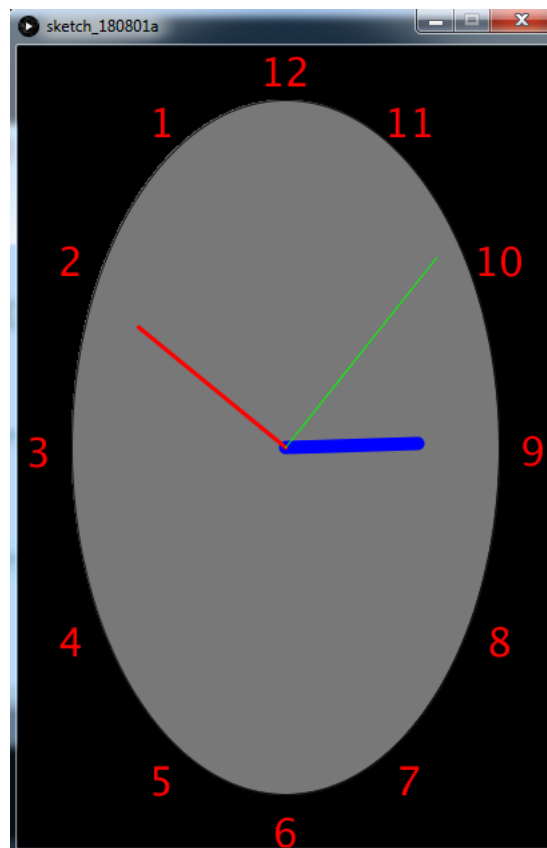
```

```

}

void draw() {
  stroke(20, 20, 20);
  fill(120, 120, 120);
  ellipse(a/2, b/2, a-80, b-80);
  fill(0, 0, 0);
  stroke(0, 0, 255);
  strokeWeight(10);
  line(a/2, b/2, (a/2*0.5)*cos(radians((hour()*5+minute()
    *1./60+15)*-6))+a/2, (b/2*0.5)*sin(radians((hour()*5+
    minute()*1./60+15)*-6))+b/2);
  stroke(255, 0, 0);
  strokeWeight(3);
  line(a/2, b/2, (a/2-80)*cos(radians((minute()+15)*-6))+a/2,
    (b/2-80)*sin(radians((minute()+15)*-6))+b/2);
  stroke(0, 255, 0);
  strokeWeight(1);
  line(a/2, b/2, (a/2-60)*cos(radians((second()+15)*-6))+a/2,
    (b/2-60)*sin(radians((second()+15)*-6))+b/2);
}

```



19. ábra. Analóg óra

4.3. Fizikai feladatok modellezése

Fizikai feladatok alatt a fizika törvényszerűségeit figyelembe vevő animációkat értünk.

4.3.1. Ütközés szimulálás

Elsőnek oldjunk meg egy egyszerű szimulációt, az ablak széleiről pattanjon vissza egy kör. A visszapattanás megvalósítása során kövessük a megfelelő fizikai törvényt. Négy eset van, bal, jobb oldal, valamint az ablak alja és teteje. Ha lerajzoljuk és kiszámoljuk, akkor a bal és jobb oldali ütközés esetén a $\beta = 2\Pi - \alpha$, míg a fenti és lenti oldallal való találkozás esetén a $\beta = \Pi - \alpha$ összefüggés adódik, ahol β az ütközés utáni, α pedig a találkozás előtti szög. Az ütközést úgy figyeljük, hogy a kör sugaránál közelebb kerül az oldal széléhez a kör.

Írjuk meg úgy a programunkat, hogy a paraméterek globális változóban helyezkedjenek el. Legyenek ezek a kör sugara, valamint a mozgás sebessége. A kör kezdeti pozícióját és indulási irányát bízzuk a véletlenre, használjuk kiszámításukhoz a random() függvényt, mely véletlen számokat generál. Minden alkalommal, amikor a függvényt meghívják, ún. véletlen értéket ad vissza a megadott tartományon belül. Ha csak egy paramétert adunk át a függvénynek, akkor egy valós (lebegőpontos) értéket ad vissza nulla és a p paraméter értéke között a $[0;p)$ balról zárt, jobbról nyílt intervallumból.

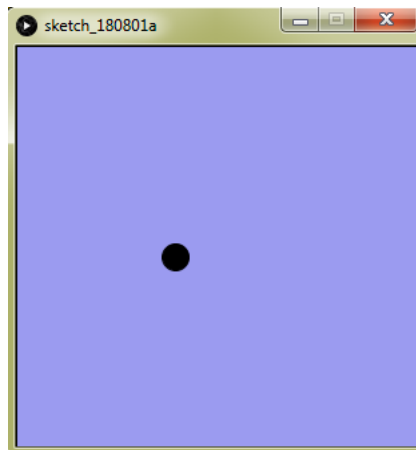
Ha két paramétert adunk meg, akkor a függvény olyan valós értéket ad vissza, amelynek értéke a két érték között van, az intervallum itt is balról zárt és jobbról nyílt. Ha egész értékre van szükségünk, akkor konvertálni kell a kapott számot. Ezek után a programunk a következő lesz, az eredményt a 20. ábra tartalmazza.

```
float r=10;
float x=random(r, 300-r);
float y=random(r, 300-r);
float v=5;
float alfa=random(TWO_PI);
void setup() {
    size(300, 300);
}

void draw() {
    fill(155, 155, 240);
    rect(0, 0, width, height);
    fill(0, 0, 0);
    ellipse(x, y, 2*r, 2*r);
    x+=v*sin(alfa);
    y+=v*cos(alfa);
    if (x<r||x>width-r)
        alfa=TWO_PI-alfa;
    if (y>height-r||y<r)
        alfa=PI-alfa;
}
```

Ettől elegánsabb, ha a kezdőértékek beállítását a setup()-ban végezzük el. A deklaráció ott nem lehetséges, mivel akkor a draw()-ból nem érénk el a változót. A draw() tehát ugyanaz marad, csak a setup() változik:

```
float r, x, y, v, alfa;
```



20. ábra. Pattogó kör

```
void setup() {
  size(300, 300);
  r=10;
  x=random(r, width-r);
  y=random(r, height-r);
  v=5;
  alfa=random(TWO_PI);
}
```

Figyeljük meg, hogy ebben az esetben x és y már az aktuális ablakmérettől függően kaphat kezdőértéket a `setup()`-ban. Valóban, ha átírjuk a `size()` paramétereit, úgy is működni fog a program.

4.3.2. Egyenletes körmozgás és harmonikus rezgőmozgás

Egyenletes körmozgásra analóg óránál már láttunk példát, bár a mozgás szakasos volt, kivéve a kismutatót. A most következő példában egy kört fogunk mozgatni egy körvonalon egyenletes sebességgel, emellett:

- Megjelenítjük az eltelt időt ezredmásodpercben:
Az idővel kapcsolatos függvényeknél már szerepelt a `millis()`, ezt fogjuk használni. Úgy érzük el, hogy pontosan az animáció kezdetétől mérje az időt, hogy felvesszünk két segéd változót: az m -be kerül a `draw()` első végrehajtásáig eltelt idő, s ebből számoljuk ki az mi változóban az animáció kezdetétől eltelt ezredmásodperceket. A gondot az okozza, hogy pl. a `setup()` végrehajtása is időbe telik, ezt kell kiküszöbölni.
- A körvonalon mozgatott kör függőleges koordinátáját használjuk a harmonikus rezgőmozgás ábrázolásához, az x koordináta fixen marad
- A harmonikus rezgőmozgás kitérés-idő grafikon szemléltetésére két megoldást mutatunk, először csak az aktuális pozíció fog kirajzolódni, másodjára láthatóvá tesszük a grafikont.

Amennyiben az egész képernyő változott, `background()`-ot használtunk, míg ha csak egy része, akkor csupán azt a részt "takarjuk el" a változások megjelenítése előtt. Az analóg óránál ez a terület a számlap volt, magukat a számokat csak egyszer rajzoltattuk ki. Ez a technika amellett,

hogy gyorsíthatja a programot, csökkenti az erőforrások igénybevételét. A mostani példában téglalapokat fogunk használni a kitakarásra, hogy a grafikont a végén láthatóvá tudjuk tenni. Az alap programunk forráskódja:

```
int m,v;
void setup()
{
    size(600, 400);
    background(0);
    textSize(32);
    m=0;
    v=2000;
}

void draw()
{
    float x,y,mi,f;
    fill(0);
    rect(0,0,320,400);
    rect(400,0,200,40);
    rect(320, 30, 600, 350);
    if (m==0)
    m=millis();
    mi=millis()-m;
    fill(255);
    text(mi/1000, width-120, 32);
    stroke(0,200,0);
    noFill();
    ellipse(160,200,300,300);
    f=(mi%v)/v;
    x=cos(f*TWO_PI)*150+160;
    y=-sin(f*TWO_PI)*150+200;
    noStroke();
    fill(200,0,0);
    ellipse((int)x,(int)y,20,20);
    stroke(0);
    fill(255,255,255);
    ellipse(400,(int)y,20,20);
}
```

Az m és mi változók szerepét már ismertettük, a v az animáció sebességét hivatott befolyásolni az f változón keresztül, melynek az értéke a $[0;1)$ intervallumban lesz mindig. A `draw()` elején a `rect()` függvényekkel takarunk, az

```
ellipse(160,200,300,300);
```

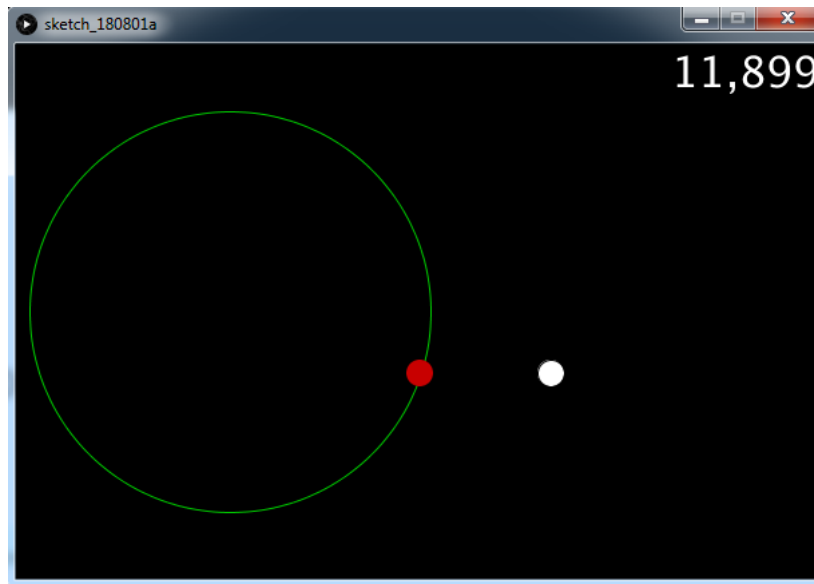
sorral rajzoltatjuk ki a referencia kört, az azon mozgó kört az

```
ellipse((int)x,(int)y,20,20);
```

sor idézi elő, s végül az

```
ellipse(400,(int)y,20,20);
```

jeleníti meg a harmonikus rezgőmozgást. Az összképet szemlélteti a 21. ábra.



21. ábra. Egyenletes körmozgás és harmonikus rezgőmozgás

Cseréljük ki az utolsó

```
ellipse(400,(int)y,20,20);
```

sorban a fix vízszintes pozíciót meghatározó 400-as értéket egy változó értékre, mely megmutatja, hogy időben hogy fut le a mozgás:

```
ellipse((int)(f*250+340),(int)y,20,20);
```

A program futtatásakor a rezgőmozgást végző körnek vízszintes irányban ekkor egyenletes sebességgel kell mozognia, erről láthatunk egy pillanatfelvételt a 22. ábrán.

Végül töröljük ki azt a téglalapot, mely letakarja a rezgőmozgást végző kör előzetes rajzát, tehát töröljük ki a

```
rect(320, 30, 600, 350);
```

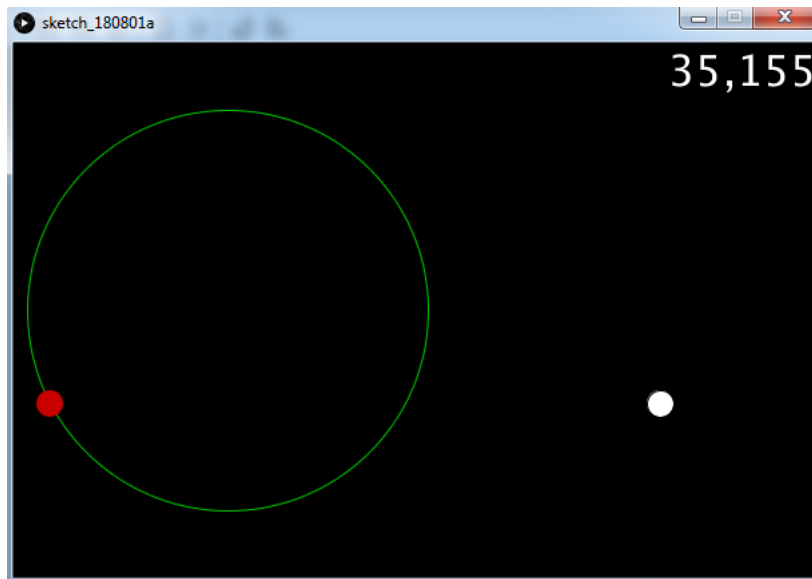
sort. Ekkor kapjuk a 23. ábrán látható képet.

4.3.3. Ferde hajítás szimulációja

A rezgőmozgáshoz hasonlóan szinte bármilyen fizikai folyamatot tudunk modellezni a Processing nyelv segítségével, aminek van vizuális megjeleníthetősége. Ebben a részben a ferde hajítást bemutató programot készítünk. A megvalósításhoz szükséges képletek:

- $x = v_0 \cdot t \cdot \cos(\alpha)$
- $y = v_0 \cdot t \cdot \sin(\alpha) - \frac{g}{2} \cdot t^2$

A program legelején határozzuk meg a paramétereiket (kezdősebesség, irányszög), valamint legyen lehetőség egy adott időpontban kiírni a koordinátáit, tehát a harmadik paraméter egy



22. ábra. Harmonikus rezgőmozgás diagram készítése



23. ábra. Harmonikus rezgőmozgás "diagram"

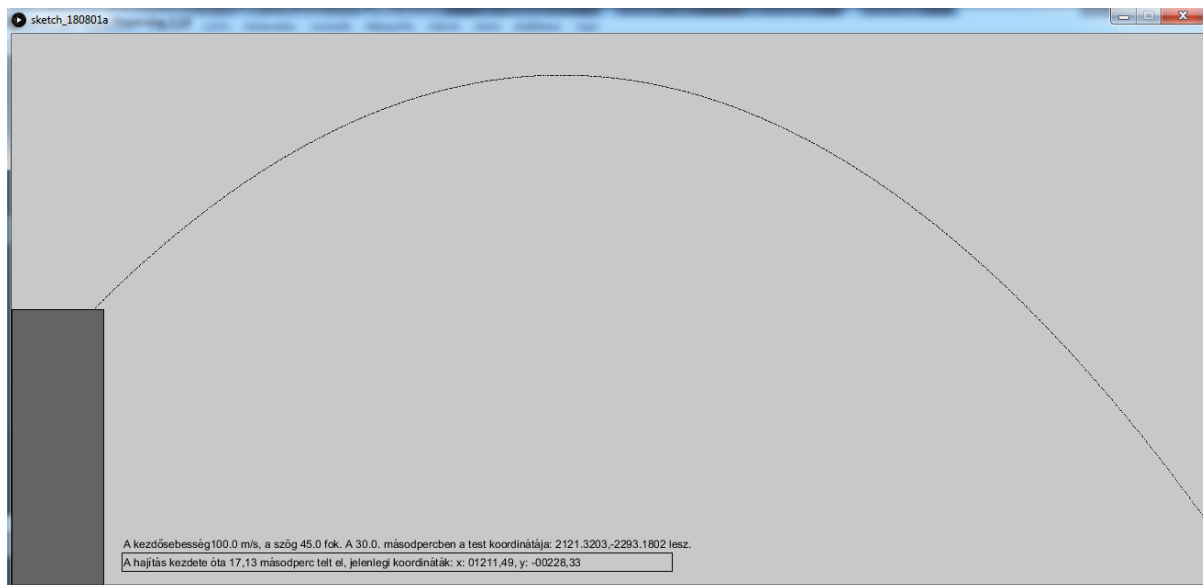
időt határozzon meg. A forrásban ezek rendre a v_0 , α , t_0 változók. Az itt szereplő $kezd$ változó a pontos mérés miatt kell, hasonlóan az előző program m változójához. Új függvény a `point()`, mellyel pontot tudunk kirajzolni. A program által kirajzolt pálya a 24. ábrán található. A program forrása:

```
float alfa=45;
float v0=100, t0=30, kezd=0;
void setup() {
  background(200);
  size(1300, 600);
  fill(100);
  rect(0, 300, 100, 300);
  fill(0);
```

```

text("A kezdosebesség"+v0+" m/s, a szög "+alfa+" fok. A "+
t0+". másodpercben a test koordinátája: "+ v0*t0*cos(
radians(alfa))+", "+(v0*t0*sin(radians(alfa))-9.81*t0*t0/2)
+" lesz.", 122, 560);
}
void draw() {
  if (kezd==0) kezd=(float)millis()/1000;
  float t=(float)millis()/1000-kezd;
  float x=v0*t*cos(radians(alfa));
  float y=v0*t*sin(radians(alfa))-9.81*t*t/2;
  fill(200);
  rect(120, 565, 600, 20);
  fill(0);
  text("A hajítás kezdete óta "+nf(t, 2, 2)+" másodperc telt
  el, jelenlegi koordináták: x: "+nf(x, 5, 2)+"", y: "+nf(y,
  5, 2), 122, 580);
  point(90+x, 300-y);
  if ((90+x>1300)||300-y<1 ) noLoop();
}

```

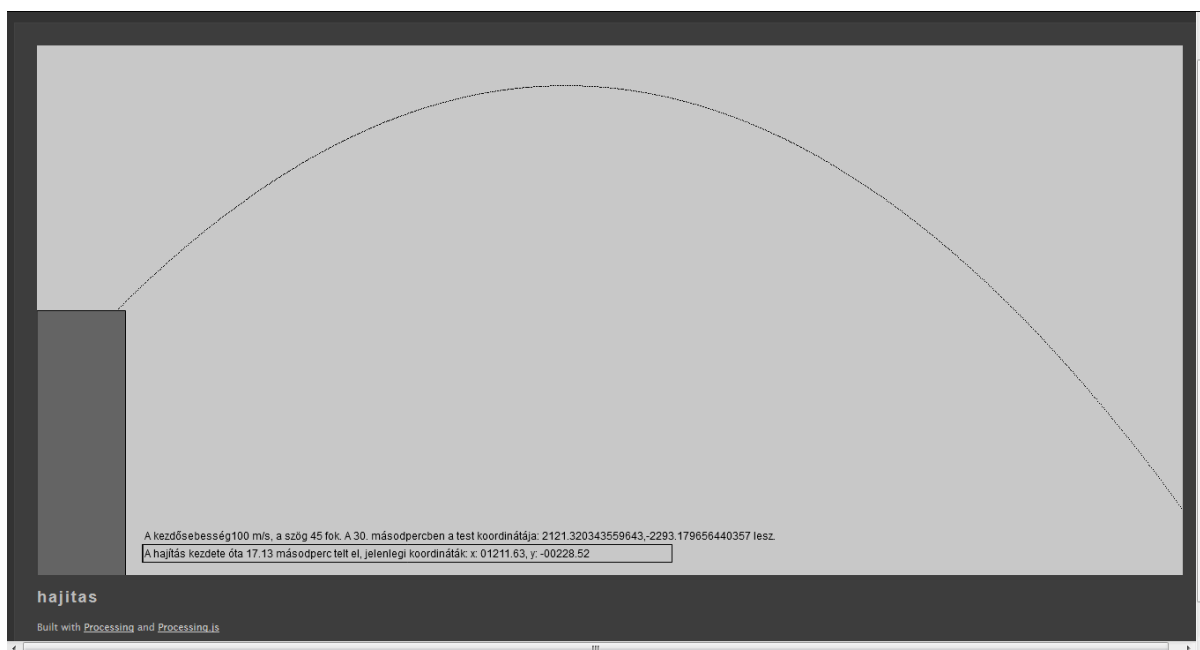


24. ábra. Ferde hajítás

A Processing némely verziójában megengedett JavaScriptre való fordítás, mely nem csak a .js kódot hozza létre, hanem a konténer weboldalt is, tehát nagyon egyszerű publikálni. A fenti program webes implementációja elérhető a <http://zeus.nyf.hu/falu/PMB2502/hajitas/> oldalon. A forrást terjedelem miatt nem illesztjük be, csak a 25. ábrán található képernyőképet, melyről látszik, hogy közel azonos az eredmény, a felbontásban van esetleg eltérés.

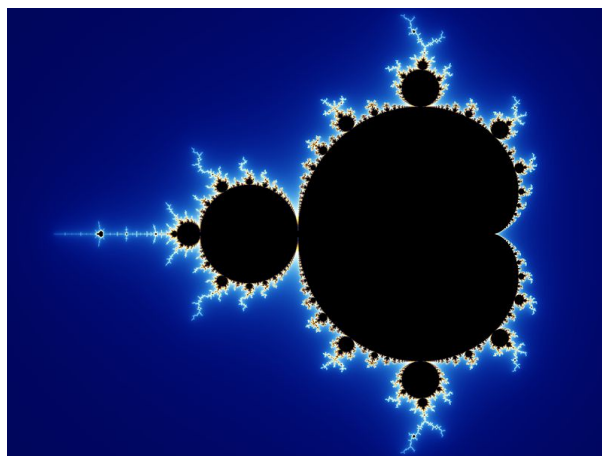
4.4. Időalapú fraktál

Bár a fizikában is megjelennek fraktálok, mégis most matematikai oldalról fogjuk megközelelni a kérdést. Nagyon sablonosan fogalmazva, a fraktál egy önhasonló alakzat. Talán a



25. ábra. Ferde hajtás megjelenítése weboldalon

leg híresebb a Mandlbrot és a Júlia halmaz. Olyan programot fogunk írni, mely időben halad-

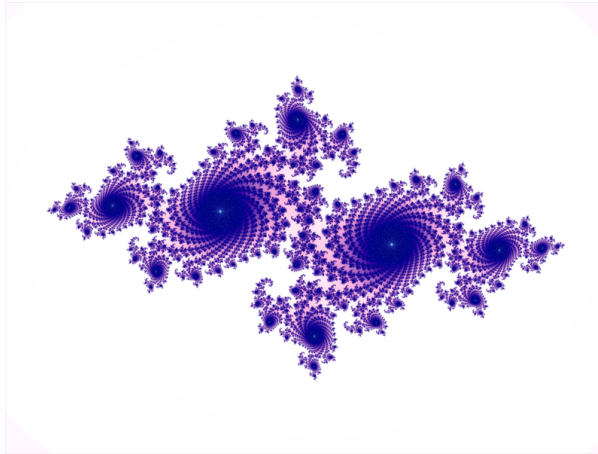


26. ábra. A Mandelbrot-halmaz. Forrás: wikipedia.org

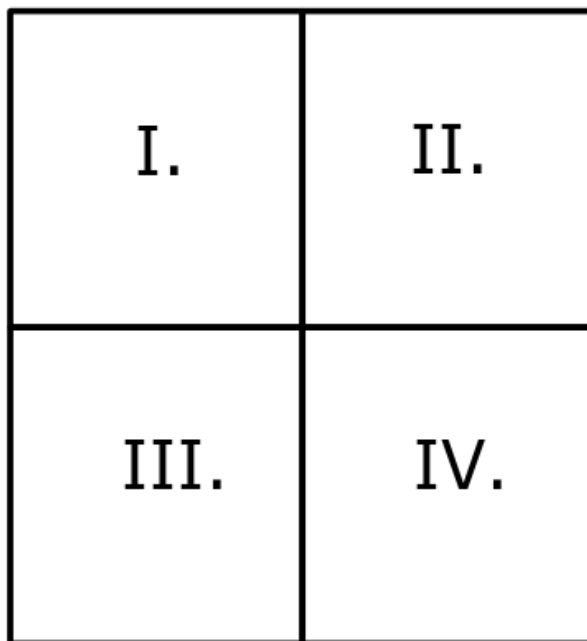
va, folyamatosan fogja megjeleníteni a fraktál pontjait. Erre az önhasonlóság ötletét használjuk fel. Vágjunk fel egy alakzatot néhány részre. Alkossunk mindegyik részalakzathoz egy-egy transzformációt, mely a teljes eredeti alakzat minden pontjához kölcsönösen egyértelműen hozzárendeli a részalakzat egy pontját. Jelöljünk ki tetszőlegesen egy pontot az alakzaton, ez lesz a megalkotandó fraktál P pontja. Véletlenszerűen válasszunk ki egy részalakzatot, alkalmazzuk a hozzá tartozó transzformációt a P pontra, a kapott P' pont az önhasonlóság miatt a fraktál pontja, így erre is alkalmazhatjuk az iménti eljárást. Tetszőleges ideig folytatható ez, s az idő haladtával egyre több pontját kapjuk meg az alakzatnak.

Az egyszerűség kedvéért négyzet legyen a kiinduló alakzatunk, s az oldalfelező merőlegesek mentén vágjuk fel négy darab egybevágó négyzetre. Mindegyik oldal hossza fele lesz a nagy négyzetnek, emiatt könnyű létrehozni a transzformációkat. Tekintsük a 28. ábrát.

Mindegyik rész esetében a transzformáció két lépésből áll. Először felére összenyomjuk a nagy



27. ábra. A Julia-halmaz. Forrás: wikipedia.org



28. ábra. A négyzet felosztása

négyzetet, majd eltoljuk a helyére. Tekintve, hogy a koordináta rendszer origója a nagy négyzet bal felső sarkában van, egyszerű a dolgunk. Az első lépés a koordináták elosztása kettővel, majd az oldalhossz felével növeljük a megfelelő koordinátákat, ha szükséges. Pontosabban:

- I. nem kell növelni
- II. x értékét kell növelni
- III. y értékét kell növelni
- IV. mind x , mind y értékét növelni kell

A kezdőpont helyzete valójában lényegtelen, "bekonvergál". Megadjuk a forrását a programnak, amit lehetne rövidebben is kódolni, de így egyértelmű a fenti eljárás alapján a kód. A `draw()`-ban a

```
for (int i=0; i<1000; i++) {
```

ciklus csak a gyorsítást szolgálja. A switch utasításról a nyelv alapjainál már volt szó. A futtatásunk eredménye egy teljesen fekete háttérű ablak, elérésének egy fázisát mutatja a 29. ábra.

```
float x=0;
float y=0;
void setup() {
    size(400, 400);
    background(255, 255, 255);
}
void draw() {
    int t;
    for (int i=0; i<1000; i++) {
        t=int(random(4));
        switch (t) {
            case 0:
                x=x/2;
                y=y/2;
                break;
            case 1:
                x=x/2+width/2;
                y=y/2;
                break;
            case 2:
                x=x/2;
                y=y/2+height/2;
                break;
            case 3:
                x=x/2+width/2;
                y=y/2+height/2;
                break;
        }
        point(x, y);
    }
}
```

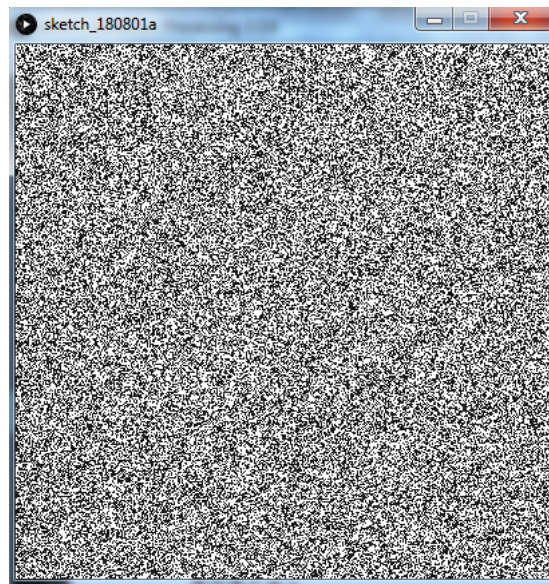
A látványos képet akkor kapjuk, ha az egyik részt kihagyjuk a generálás során, pl. a választásnál

```
t=int(random(3));
```

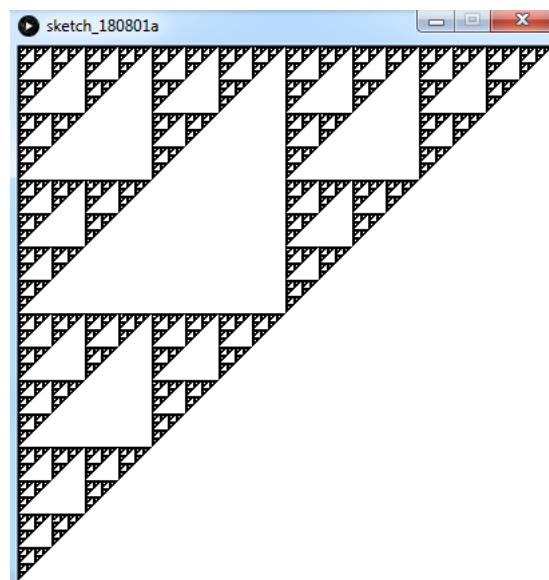
kerül be, vagy egyszerűen kitöröljük valamelyik case ágat. Ezt szemlélteti a 30. ábra.

A kapott alakzatot Sierpinski háromszögnek nevezik, de elterjedtebb az "álló" helyzetű változata. Tekintsük meg az ezt kirajzoló programot, itt már éltünk a rövidebb kód lehetőségével. Az eredményt a 31. ábrán láthatjuk.

```
void setup() {
    size(600, 400);
    background(0, 0, 0);
```



29. ábra. Fáziskép



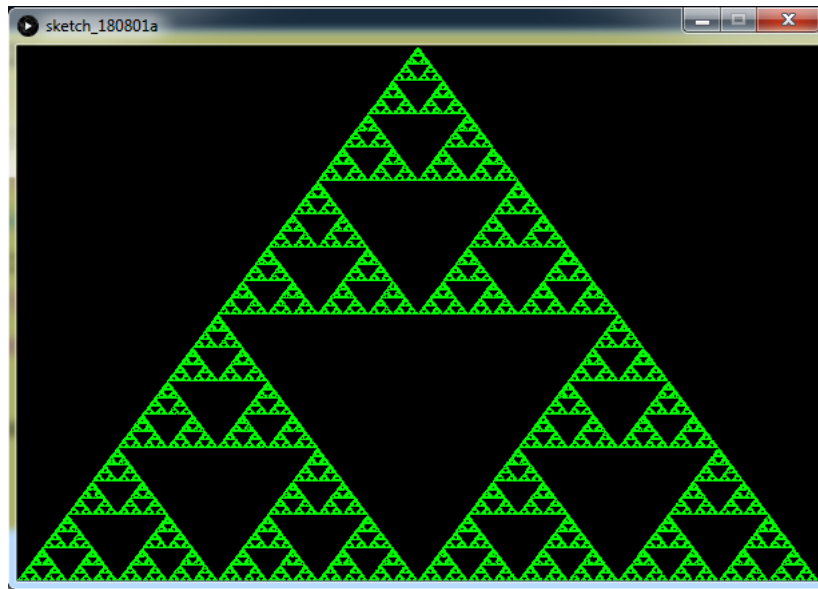
30. ábra. Elhagyva a IV. alakzatrészt

```
stroke(0, 255, 0);
}
int x, y;
void draw() {
  for (int i=0; i<200; i++) {
    x=x/2;
    y=y/2;
    switch (int(random(3))) {
      case 1:
        x+=width/2;
        break;
      case 2:
```

```

    x+=width/4;
    y+=height/2;
  }
  point(x, height-y);
}
}

```



31. ábra. Sierpinski háromszög

A fentiek alapján különösebb magyarázat nélkül megadjuk a Sierpinski szőnyeget rajzoló forráskódot és a 32. ábrán a program eredményét.

```

void setup() {
  size(600, 600);
  background(0, 0, 0);
}
int x, y;
void draw() {

  stroke(0, 255, 0);
  for (int i=0; i<1000; i++) {
    switch (int(random(9))) {
      case 0:
        x=x/3;
        y=y/3;
        break;
      case 1:
        x=x/3+width/3;
        y=y/3;
        break;
      case 2:
        x=x/3+2*width/3;
        y=y/3;

```

```

        break;
        case 3:
            x=x/3;
            y=y/3+height/3;
            break;
        case 4:
            //x=x/3+width/3;
            //y=y/3+height/3;
            break;
        case 5:
            x=x/3+2*width/3;
            y=y/3+height/3;
            break;
        case 6:
            x=x/3;
            y=y/3+2*height/3;
            break;
        case 7:
            x=x/3+width/3;
            y=y/3+2*height/3;
            break;
        case 8:
            x=x/3+2*width/3;
            y=y/3+2*height/3;
            break;
    }

    point(x, y);
}
}

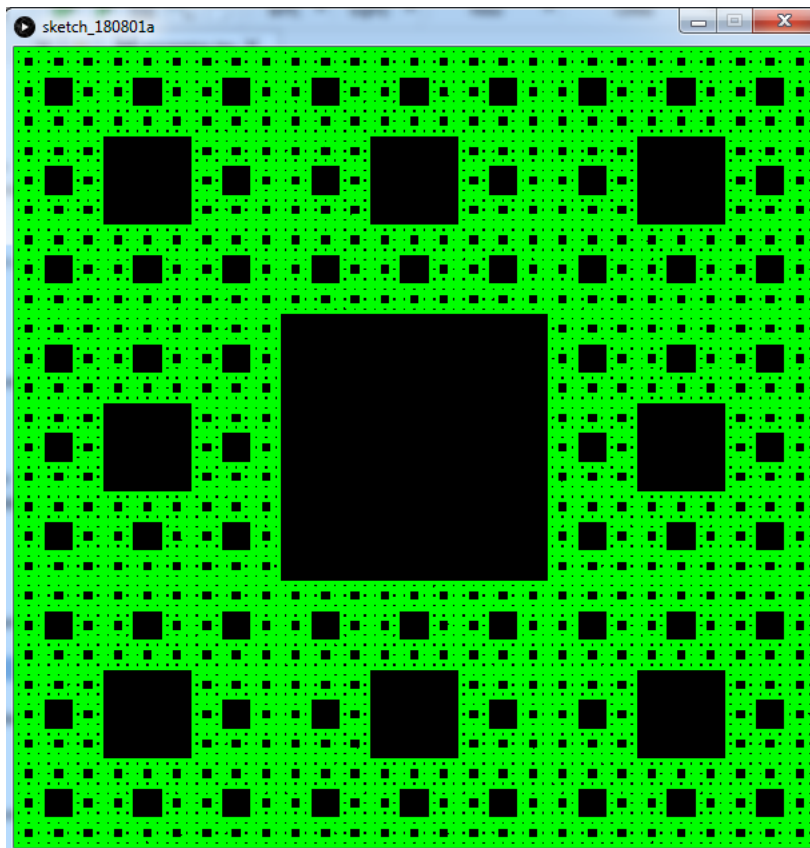
```

4.5. Fényűjság

Írjunk programot, mely adott idő alatt elvégez egy feladatot, majd leáll. Jelen esetben a feladat egy adott szöveg adott idő alatt való végigfuttatása az ablakban vízszintesen. Működjön ablakmérettől függetlenül, paraméterekben tudjuk szabályozni a szöveg tartalmát, az időt és a betűk nagyságát. A forrásban emiatt az első két sorban lévő változóknak adjuk meg ezeket:

- `meret`: a betűk mérete
- `ido`: ezredmásodpercben a végigfutás ideje
- `s`: maga a szöveg tartalma

A program nem fog pontosan működni, de elég pontos az elvárásokhoz képest. A pontatlanságnak a digitális megjelenítés és számolás az oka. A programban használni fogjuk a `String` osztályt, ezzel a típussal tudunk szöveget tárolni. A szöveg hosszával nem kell foglalkozni, mivel alapértelmezetten balra van igazítva a kiírás. A forrásprogram a következő lesz, valamint a 33. ábrán látható egy képernyőkép is. Az ellenőrzés miatt kiíratjuk az eltelt időt. A program lényege az



32. ábra. Sierpinski szőnyeg

```
i=x*width/ido;
```

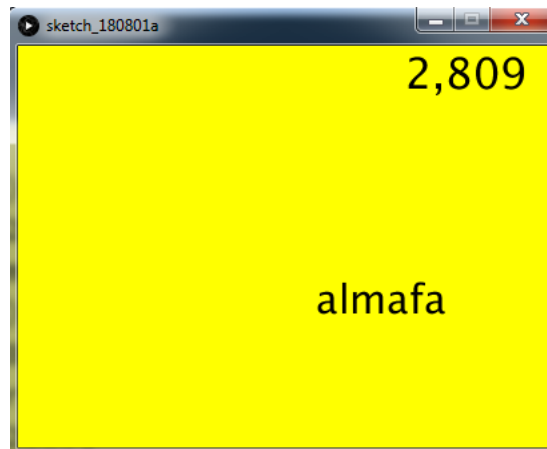
sorban van, ahol x az eltelt időt tartalmazza.

```
int i,m,meret=30,ido=5000,x;
String s="almafa";
void setup()
{
  size(400, 300);
  fill(0);
  m=i=0;
}
void draw()
{
  if (m==0)
  m=millis();
  background(255, 255, 0);
  x=millis()-m;
  textSize(32);
  text(1.*x/1000, width-120, 32);
  textSize(meret);
  text(s, i, 200);
  i=x*width/ido;
  if (i>width) {
```

```

    noLoop();
  }
}

```



33. ábra. Szöveg végigfuttatása

Hasonlóan működik a következő program, melyben kirajzoltatjuk a szivárvány színeit adott idő alatt. Újdonság a HSB mód, amit a

```
colorMode(HSB,400);
```

sorral állítunk be. Ez azért előnyös, mert így a fill() függvény első paramétere tartalmazza a színkódot, nem kell foglalkozni azzal, hogy miként kell kikeverni egymás után a szivárvány színeit az RGB komponensekből. A forrás programot közöljük csak, futtatáskor egy ablakot látunk, melynek színe az adott idő alatt veszi fel a szivárvány színeit, ez itt most épp 3 másodperc.

```

int ido=3000,rido;
void setup(){
  size(800,600);
  colorMode(HSB,400);
  rido=0;
}
void draw(){
  if (rido==0)rido=millis();
  float akt=(millis()-rido)%ido;
  akt=akt*399/(ido-1);
  fill(akt,400,400);
  rect(100,50,width-200,height-100);
}

```

5. Összetett időfüggő feladatok kódolása Processing környezetben

Összetett feladatnak tekintjük azokat a problémákat, melyek az alapismereteken túli anyagot tartalmaznak. Ebben a fejezetben ismertetésre kerül a saját osztály használata, valamint a program futási időben való vezérlése felhasználói inputokkal, tehát a billentyűzet- és az egérkezelés. Ezekkel a technológiákkal interaktív programokat hozhatunk létre, ami nagyban megnöveli a használhatóságukat és a felhasználói élményt.

Eddig csak annyiban jelent meg ez a jellemző, hogy globális változókkal a program elején újabb és újabb teszteléseknek vetettük alá az általunk írt alkalmazást, ami nem túl elegáns dolog. Teljesen egyenértékű lett volna azzal, hogy akárhol módosítjuk a program kódot, csupán kényelmi szempontból jó, hogy a program elején helyezkednek el ezek az átírható értékek. A továbbiakban szintet lépünk, egy forráskódot használva a felhasználói beavatkozásoktól függően más-más futási eredményt fogunk elérni. Elsőnek egy ezt mellőző példát oldunk meg, ahol viszont megismerkedhetünk a többes esemény egy lehetséges lekódolásával, amit saját osztály segítségével viszünk véghez. A 4.3.1. fejezetben megoldott problémát fogjuk tovább vinni:

Jelenítsünk meg öt másodpercenként egy ablakban mozgó köröket, melyek az ablak széléről visszapattannak (egymásról nem). A megjelenő köröknek legyen véletlen a nagysága, kiinduló helyzete, a kiindulás iránya, valamint gyorsuljon az elindulással ellentétes irányban. Egy lehetséges megoldás a következő:

```
import java.util.ArrayList;
float epsilon=0.02;
ArrayList<Golyo> l=new ArrayList<Golyo>();

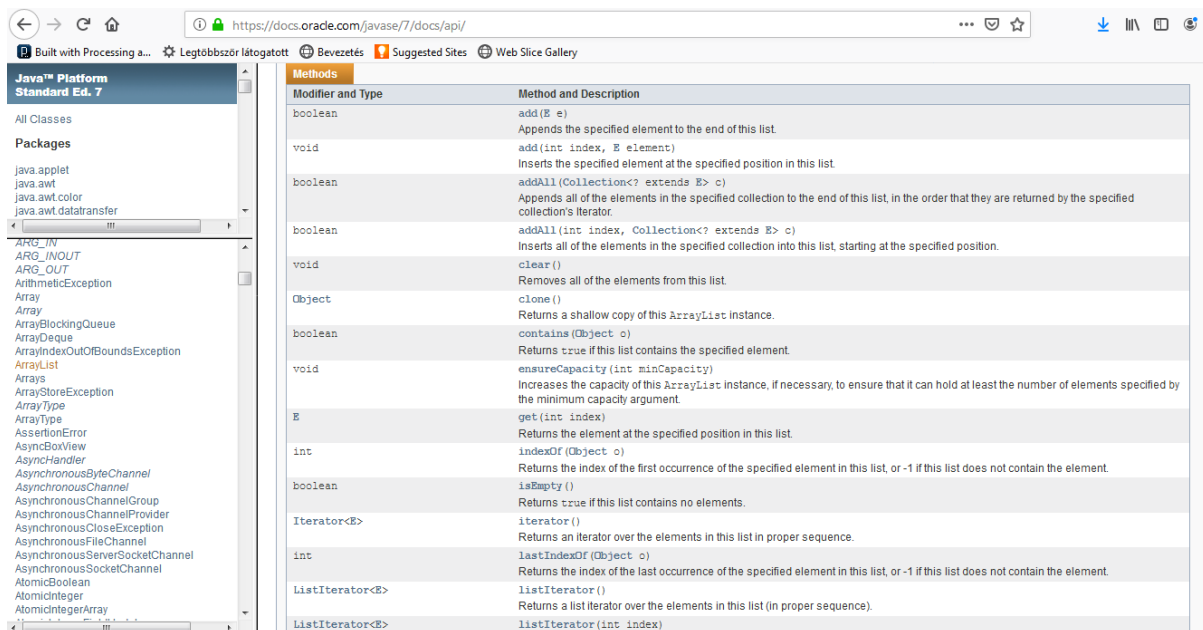
int regimillis=0;
void setup() {
    size(300, 300);
}
class Golyo {
    float x, y, v, r, alfa;
    Golyo() {
        r= random(20,30);
        x=random(r, 300-r);
        y=random(r, 300-r);
        v=12;
        alfa=random(TWO_PI);
    }
    void update() {
        if (x<r||x>width-r)
            alfa=TWO_PI-alfa;
        if (y>height-r||y<r) alfa=PI-alfa;
        v-=epsilon;
        x+=v*sin(alfa);
        y+=v*cos(alfa);
        fill(0, 100, 0);
        ellipse(x, y, 2*r, 2*r);
    }
}
```

```

}
void draw() {
    if (millis()/1000>regimillis) {
        regimillis+=5;
        Golyo g=new Golyo();
        l.add(g);
    }
    fill(100, 0, 0);
    rect(0, 0, width, height);
    for (Golyo i : l) {
        i.update();
    }
}
}

```

Az osztály és objektum kezelésén kívül nincs benne újdonság. Az ötlet az, hogy öt másodpercenként adjunk hozzá egy gyűjteményhez egy kört, valamint minden egyes draw() végrehajtásnál a gyűjtemény minden egyes elemének aktualizáljuk a pozícióját és rajzoltassuk ki. A gyűjtemény megvalósítása az ArrayList osztállyal fog történni, ami a List interfész egy implementációja (több interfészt is implementál, de erre nem térünk ki). Számunkra azért lesz előnyös, mert korlátozás nélkül, és egyszerűen tudunk elhelyezni benne objektumot, és el tudjuk érni ezeket anélkül, hogy tudnánk, hány darab van benne.



34. ábra. Az ArrayList osztály referencia lapja

Az ArrayList alapesetben nem érhető el, emiatt szerepel az

```
import java.util.ArrayList;
```

sor a program legelején. Az importálás után már használhatjuk, létrehozunk az osztályból egy objektumot, egy konkrét gyűjteményt, melyre az *l* azonosítóval tudunk majd hivatkozni a későbbiekben:

```
ArrayList<Golyo> l=new ArrayList<Golyo>();
```

Létrehozunk egy saját osztályt Golyo néven, melyben öt változó és két metódus foglal helyet első ránézésre:

```
class Golyo {  
    float x, y, v, r, alfa;  
    Golyo() {  
        .....  
    }  
    void update() {  
        .....  
    }  
}
```

A változók példányváltozók, az osztályból létrehozott minden egyes objektumban ezek különböző értéket vehetnek fel. A Golyo() nem függvény lesz, nincs visszatérési értéke, valamint a neve megegyezik az osztály nevével, innen fel lehet ismerni, hogy konstruktorról van szó. Valahányszor példányosítani fogjuk az osztály, azaz létrehozunk belőle egy objektumot, a konstruktorban megadott utasítások mind végrehajtnak:

```
r=random(20,30);  
x=random(r, 300-r);  
y=random(r, 300-r);  
v=12;  
alfa=random(TWO_PI);
```

Látható, hogy véletlen értéket kap az r, x, y, alfa adattagja az objektumnak és a v értéke 12 lesz. Az update() viszont valódi metódus, mindegyik objektum rendelkezni fog vele. A benne lévő sorok ismerősek a korábbi programból, ezzel ismerjük fel és kezeljük az ütközéseket az ablak szélénél. Egyedül a

```
v -= epsilon;
```

sor különbözik, ez felel a lassulásért. A draw() metódus elején az

```
if (millis()/1000>regimillis) {  
    regimillis+=5;
```

programrészlettel érjük el, hogy 5 másodpercenként hajtsdjon végre az igaz ágban lévő blokk-utasítás, míg a

```
Golyo g=new Golyo();
```

sor létrehoz egy új pattogó kört a fentebb említett módon, és az

```
l.add(g);
```

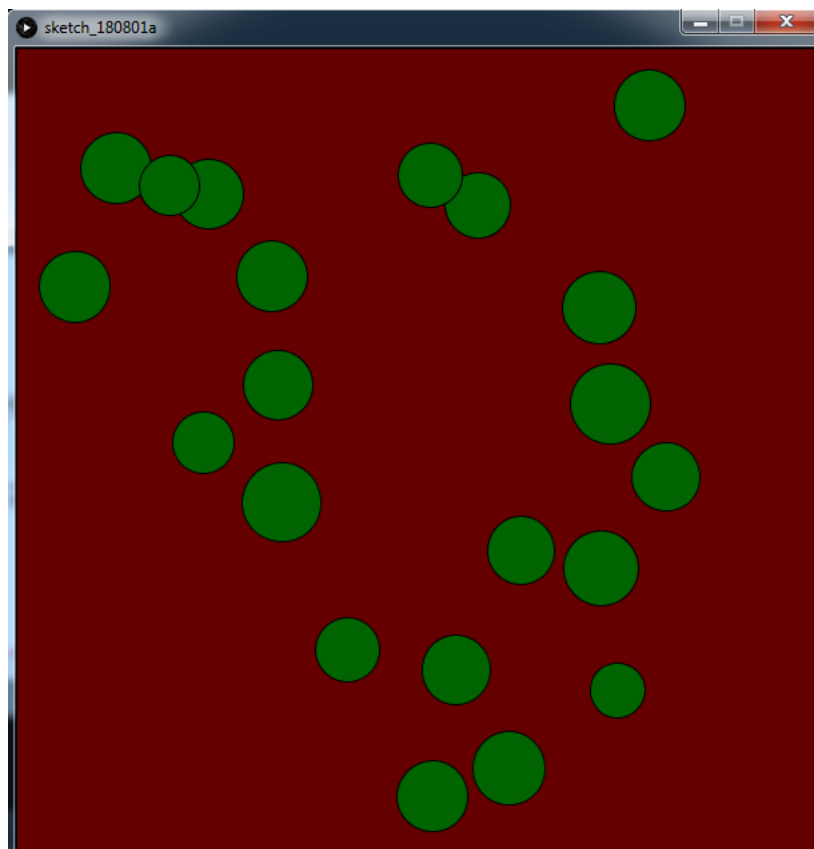
biztosítja, hogy bekerüljön az l gyűjteménybe. A

```
fill(100, 0, 0);  
rect(0, 0, width, height);
```

biztosítja a törlést, használhattunk volna `background()` függvényt is helyette, a körök kirajzolásáért pedig a

```
for (Golyo i : l) {  
    i.update();  
}
```

felel. A `for` ciklus ezen változata végig járja az `l` gyűjtemény minden `-Golyo` osztályból származó- elemét, s a `Golyo` osztályból származtatott `i` objektum felveszi egyenként a bejárt objektumok referenciáját. Így már meghívható az `update()` metódusuk, ami a fentiek alapján lekezeli az ütközést és a kirajzolást. A program egy pillanatképe a 35. ábrán van.



35. ábra. Több kör animációja ütközéssel

5.1. Billentyűzet kezelése

A felhasználó és a programok kapcsolatát az egér és a billentyűzet biztosítja leggyakrabban. Processing nyelven is lehetőségünk van ezek kezelésére, de ha csak a Processingre támaszkodunk, akkor ne várjunk túl nagy lehetőségeket a billentyűzet tekintetében. A következő elemeket tartalmazza a referencia a Keyboard szekcióban:

- `key`: A legutóbb használt (lenyomott, vagy felengedett) billentyű kódja. Hasonlóan, mint a `width` vagy `height` ún. automatikus, vagy más néven rendszerváltozó. Csak olvasásra szolgál, értékét nem állítjuk. Nem ASCII kódú karakterek esetén használja a `keyCode`

változót. Az ASCII specifikációban szereplő karaktereknél (BACKSPACE, TAB, ENTER, RETURN, ESC és DELETE) nem kell ellenőrizni, hogy több karakteres-e, azaz kódolt-e.

- `keyCode`: A `keyCode` speciális gombok, például a kurzormozgató billentyűk (UP, DOWN, LEFT, RIGHT), valamint az ALT, CONTROL és SHIFT felismerésére szolgál. Ezek ellenőrzésekor hasznos lehet először azt megnézni, hogy kódolt-e. Ez az if utasítással tehető meg, ahol a feltétel (`key == CODED`). Ha kódolt, akkor lehet felismerni a felsorolt konstansokkal.
- `keyPressed()`: A `keyPressed()` ún. **eseményfüggvény** -amennyiben szerepel- minden gombnyomás esetén meghívásra kerül. Gondot okozhat, hogy egy gomb lenyomva tartása több hívást vált ki. Az eseményfüggvényeket a `setup()` és `draw()` függvényhez hasonlóan nem hívjuk meg, automatikus a végrehajtásuk. Az egér és a billentyűzet eseményfüggvényei csak akkor működnek, ha van `draw()` függvény a programban. A `draw()` nélkül a kódot csak egyszer futtatják, majd abbahagyják az események figyelését.
- `keyPressed`: A rendszerváltozó igaz, ha bármelyik gombot megnyomják, és hamis, ha egyetlen gombot sem nyomnak meg.
- `keyReleased()`: Eseményfüggvény, gomb felengedésekor hívódik meg.
- `keyTyped()`: Hasonló a `keyPressed()` eseményfüggvényhez, de nem hívódik meg ALT, CONTROL és SHIFT esetén.

5.1.1. Szám tippelés

Demonstrációképpen tekintsük a következő programot: Sorsoljunk ki egy számot, a program tegye lehetővé, hogy két játékos felváltva tippelhessen a kitalált számra. Mindegyik játékosnak folyamatosan írja ki a gondolkozásra fordított idejét, sikeres tipp esetén írja ki, hogy "Talált!!". Az ötlet az, hogy kisorsoljuk a számot, egy változóban nyilvántartjuk, ahogy azt is, hogy melyik játékos jön. A beolvasást az imént ismertetett `keyReleased()` eseményfüggvénnyel oldjuk meg, ez kezeli a beolvasást, valamint a beolvasás utáni tipp vizsgálatot, a játékosok cseréjét. A `setup()`-ban az idők kiírása történik, tehát a grafikus megjelenítés hajtódik végre:

```
int t[]={0, 0}, x, tipp, i=0, rm;
color c[]={
  color(255, 0, 0),
  color(100, 0, 0),
  color(0, 0, 100),
  color(0, 0, 255)};

void setup() {
  size(650, 300);
  background(0);
  x=int(random(1, 100));
  textSize(20);
  rm=millis();
}

void draw() {
  t[i]+=millis()-rm;
```

```

rm=millis();
background(0);
fill(255);
text(x,300,50);
fill(c[i]);
text(t[0]/1000+" ":"+t[0]%1000/10, 40, 270);
ellipse(40, 200, 40, 40);
fill(c[i+2]);
text(t[1]/1000+" ":"+t[1]%1000/10, 560, 270);
ellipse(560, 200, 40, 40);
}

void keyReleased() {
  if (key== ENTER) {
    if(tipp==x){
      textAlign(CENTER,CENTER);
      fill(c[i*3]);
      textSize(40);
      text("áTallt!!",width/2,height/2);
      noLoop();
    }
    i=++i%2;
    tipp=0;
  } else {
    tipp=tipp*10+key-'0';
    println(tipp);
  }
}

```



36. ábra. Szám tippelés időméréssel

A programban megjelenő új eszköz a tömbhasználat, illetve a color típus, amit a nyelvi elemeknél már említettünk. A következő sorban szerepel a két db. int típust tartalmazó *t* tömb létrehozása és feltöltése nullákkal.

```
int t[]={0, 0}
```

Hasonlóan történik a *c* létrehozása, újdonság a `color()` függvény, mellyel arra mutatunk példát, hogy RGB kód segítségével hogyan adjunk értéket egy `color` típusú változónak:

```
color c[]={
    color(255, 0, 0),
    color(100, 0, 0),
    color(0, 0, 100),
    color(0, 0, 255)};
```

A globális változók szerepei:

- `t[]`: A játékosok idejének nyilvántartása
- `x`: A kisorsolt szám
- `tipp`: Az aktuális játékos tippje
- `i`: Az aktuális játékos sorszáma (0, vagy 1)
- `c[]`: A színek nyilvántartása:
 0. Az első játékos aktív színe
 1. Az első játékos passzív színe
 2. A második játékos passzív színe
 3. A második játékos aktív színe

A `keyReleased()` függvényben az esetleg nehezebben megérthető részek: a

```
fill(c[i*3]);
```

sorban *i* értékétől függően a *c*-ből valamelyik aktív szín választódik ki. Az

```
i=++i%2;
```

sor biztosítja a játékosok váltását, ha *i* egy volt, akkor nulla lesz és viszont. A

```
tipp=tipp*10+key - '0';
println(tipp);
```

programrészlet első sora rakja össze a beolvasott számjegyekből a számot. A `key-'0'` segítségével készítünk számot a karakterből, a `println()` itt csak a program futásának nyomon követésére szolgál a tesztelés során. Az ablak csak akkor tud eseményt fogadni, ha nála van a fókusz, emiatt ne felejtünk belekattintani, vagy egyéb módon odaadni a fókust, különben az esemény, pl. a számok begépelése nem a program ablakában fog eseményt kiváltani, s emiatt nem lesz hatással a program működésére.

5.2. Az egér kezelése

A billentyűzet mellett a legfontosabb beviteli eszköz az egér, ennek kezelését ismertetjük ebben a fejezetben. A billentyűzet kezeléséhez hasonló technológia áll rendelkezésre, lehetőségünk nyílik mind rendszerváltozókon, mind eseményfüggvényeken keresztül hozzáférni az egér különböző állapotaihoz, értve ez alatt a gombokat, és a pozíciót. A referencia oldalon a Mouse címszó alatt találjuk meg az alábbi lehetőségeket:

- `mouseButton`: A rendszerváltozó `LEFT`, `RIGHT`, vagy `CENTER` értéket vesz fel a lenyomott gombtól függően. Érdeemes a vizsgálata előtt ellenőrizni, hogy történt-e gombnyomás.
- `mouseClicked()`: A függvény egy egérgomb lenyomása és elengedése után hívódik meg. Akárcsak a billentyűzet kezelésénél, az eseményfüggvények csak akkor működnek, ha van `draw()`, különben a kódot csak egyszer futtatják le, majd abbahagyják az események figyelését.
- `mouseDragged()`: Vonszoláskor kerül meghívásra, azaz ha lenyomott gombbal mozgatjuk az egeret.
- `mouseMoved()`: Egérmozgatásánál aktivizálódik.
- `mousePressed()`: Gomb megnyomásának eseménye. A `mouseButton` rendszerváltozóból olvasható ki a lenyomott gomb.
- `mousePressed`: Rendszerváltozó, visszaadja logikai értéként, hogy volt-e lenyomva gomb.
- `mouseReleased()`: Gomb felengedésének eseménye
- `mouseWheel()`: A `mouseWheel()` eseményfüggvény akkor kerül meghívásra, amikor az egér görgőjét mozgatjuk. (Néhány egérnek nincs, természetesen ez a funkció csak azokra az egerekre vonatkozik, amelyeknek van.) A függvényben használt `getCount()` függvény pozitív értékeket ad vissza, amikor a felhasználó felé mozdul a görgő, s negatívát ellenkező esetben. Referencia példa:

```
void mouseWheel(MouseEvent event) {  
    float e = event.getCount();  
    println(e);  
}
```

- `mouseX`: Az egér x koordinátáját tartalmazó rendszerváltozó, tehát értéket automatikusan kap
- `mouseY`: Az egér y koordinátáját tartalmazó rendszerváltozó
- `pmouseX`: A `pmouseX` rendszerváltozó mindig az egér vízszintes helyzetét tartalmazza az aktuális képkockát megelőző képkockához képest. Előfordulhat, hogy a `pmouseX` és a `pmouseY` eltérő értékeket mutat abban az esetben, ha a `draw()`-ban és valamely eseményfüggvényben kerül meghívásra. A `draw()` függvényen belül a `pmouseX` és a `pmouseY`

képkockánként csak egyszer frissül automatikusan, ahogy az eseményfüggvényeken belül is, azonban egy esemény akár többször is megtörténhet egy draw() alatt. Ha az értékeket nem frissítették azonnal az egér eseményei során a draw()-ban, akkor az egér pozícióját képkockánként csak egyszer olvassa le. A frissítés megtörténik a draw() függvényen belül mindig, amikor a pmouseX, pmouseY, mouseX, mouseY változókra hivatkozunk. Innen adódik, hogy ha az előző képkockához viszonyítva szeretne értékeket használni, akkor a draw() függvényben használja a pmouseX és pmouseY rendszerváltozókat, ha pedig folyamatos visszajelzésre van szükség, akkor az egér eseményfüggvényében érdemes alkalmazni.

- pmouseY: az egér függőleges helyzetét tartalmazza az aktuális képkockát megelőző képkockához képest.

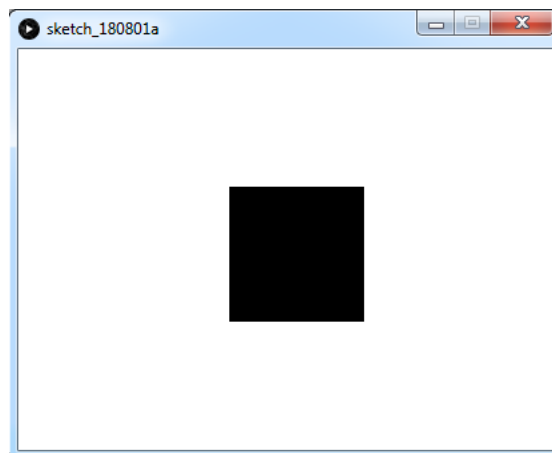
5.2.1. Négyzet vonszolása

Elsőnek nézzünk egy egyszerű feladatot, amely során egy 100x100-as négyzetet tudunk vonszolni az egérrel egy tetszőleges méretű ablakban. A program korlátozza le a négyzet mozgását, azaz mindig teljes terjedelmében az ablakban látszódjon. A program forráskódja:

```
int x=0, y=0, deltax, deltay;
boolean vonszol=false;
void setup() {
  size(400, 300);
  textSize(50);
}
void draw() {
  x=constrain(x, 0, width-101);
  y=constrain(y, 0, height-101);
  background(255);
  fill(0);
  rect(x, y, 100, 100);
}
void mouseDragged()
{
  if (vonszol) {
    x= mouseX-deltax;
    y= mouseY-deltay;
  }
}
void mousePressed()
{
  if (mouseX-x<100&&mouseX-x>0 && mouseY-y<100&&mouseY-y>0) {
    vonszol=true;
    deltax=mouseX-x;
    deltay=mouseY-y;
  }
  else
    vonszol=false;
}
```

```
}
```

A 37. ábrán a program futási eredménye látható. A program túl sok magyarázatot nem igényel,



37. ábra. Négyzet vonszolása

a változóknban a következő értékeket tároljuk:

- x: a négyzet vízszintes pozíciója
- y: a négyzet függőleges pozíciója
- deltax: a vonszolás megkezdésekor a négyzet bal felső sarkának és az egér pozíciójának vízszintes irányú különbsége
- deltay: a vonszolás megkezdésekor a négyzet bal felső sarkának és az egér pozíciójának függőleges irányú különbsége
- vonszol: történjen-e vonszolás

A négyzet ablakban maradását a

```
x=constrain(x, 0, width-101);  
y=constrain(y, 0, height-101);
```

sorok biztosítják. Az

```
if (mouseX - x < 100 && mouseX - x > 0 && mouseY - y < 100 && mouseY - y > 0) {
```

sor miatt csak a négyzetbe való kattintásnál fog aktív tevékenységet kifejteni a `mouseDragged()` eseményfüggvény, mivel a `vonsszol` csak itt kaphat igaz értéket.

5.2.2. Szakasz rajzolása

Írjunk programot, mellyel színes szakaszokat tudunk rajzolni egér segítségével. Legyen lehetőség egérrel történő színválasztásra, az aktuális szín kerüljön kijelzésre. A szakasz rajzolása a kezdőpont, majd a végpont kijelölésével történjen meg. Egy lehetséges implementációt mutatunk be az alábbiakban. A program forráskódját az alább adjuk meg, a program futásának eredménye a 38. ábrán tekinthető meg.

```

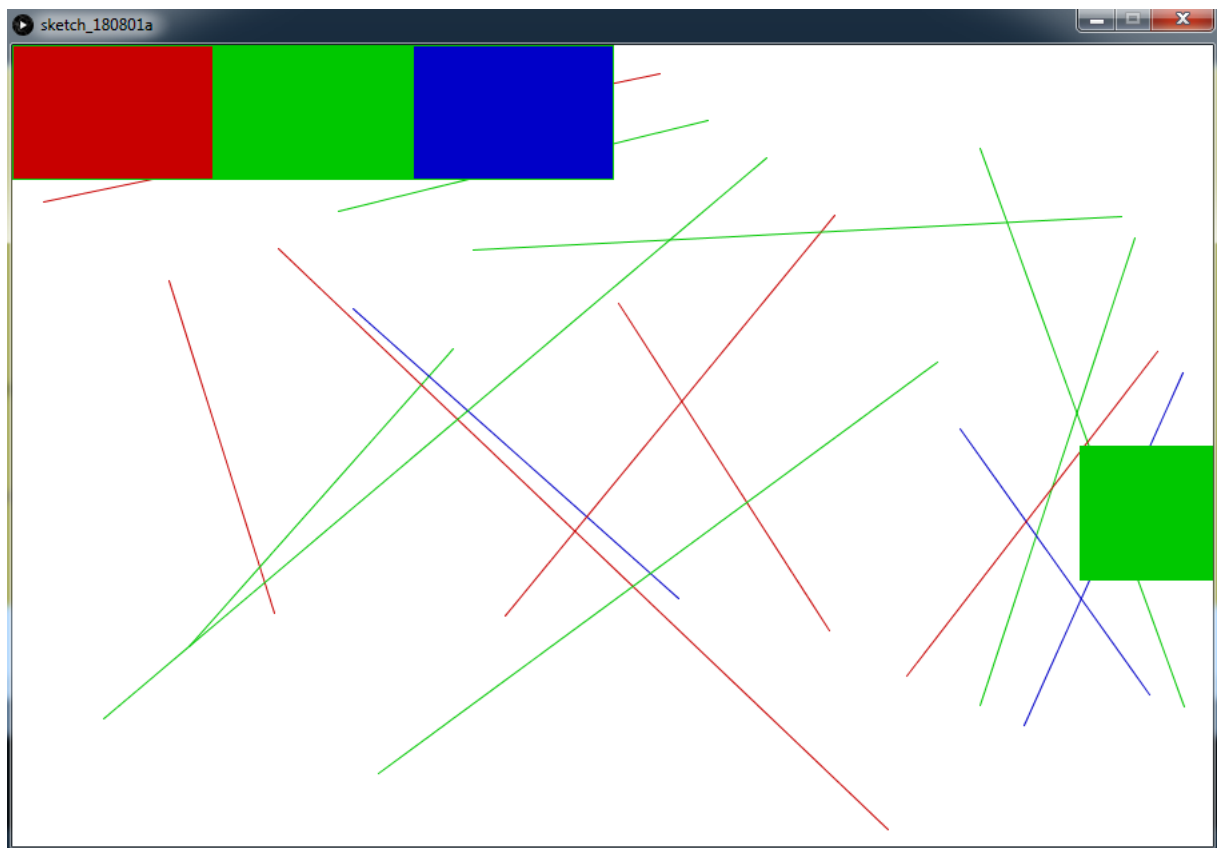
int x1, y1, x2, y2;
color c;
boolean kezd=true;
void setup() {
  size(900, 600);
  background(255);
  c=color(200, 0, 0);
}
void draw() {
  fill(200, 0, 0);
  rect(0, 0, 150, 100);
  fill(0, 200, 0);
  rect(150, 0, 300, 100);
  fill(0, 0, 200);
  rect(300, 0, 450, 100);
  fill(c);
  rect(800, 300, 900, 400);
}
void mousePressed() {
  if (mouseX<=450 && mouseY<100)
    c = get(mouseX, mouseY);
  else {
    if (kezd) {
      x1=mouseX;
      y1=mouseY;
      kezd=false;
    }
    else {
      x2=mouseX;
      y2=mouseY;
      stroke(c);
      line(x1, y1, x2, y2);
      kezd=true;
    }
    println(kezd+"x1="+x1+"y1="+y1+"x2="+x2+"y2="+y2);
  }
}

```

A programban alkalmazott változók és használatuk:

- x1, y1, x2, y2: a rajzolandó szakasz kezdő- és végpontjának koordinátái
- c: a kiválasztott szín
- kezd: kezdőpont/végpont meghatározása. Ha igaz, akkor kezdőpont, ha hamis, akkor végpont.

A draw() metódusban kirajzoljuk a színválasztást lehetővé tévő három téglalapot, valamint az aktuális színt jelző négyzetet. A setup() függvényben a szokásos beállítások mellett kezdő színnek a pirosat adjuk meg. A draw()



38. ábra. Szakasz rajzoló program felülete

```
fill(200, 0, 0);
rect(0, 0, 150, 100);
fill(0, 200, 0);
rect(150, 0, 150, 100);
fill(0, 0, 200);
rect(300, 0, 150, 100);
```

sorai biztosítják a fentre kerülő színválasztást lehetővé tevő téglalapok megrajzolását, míg a

```
fill(c);
rect(800, 300, 100, 100);
```

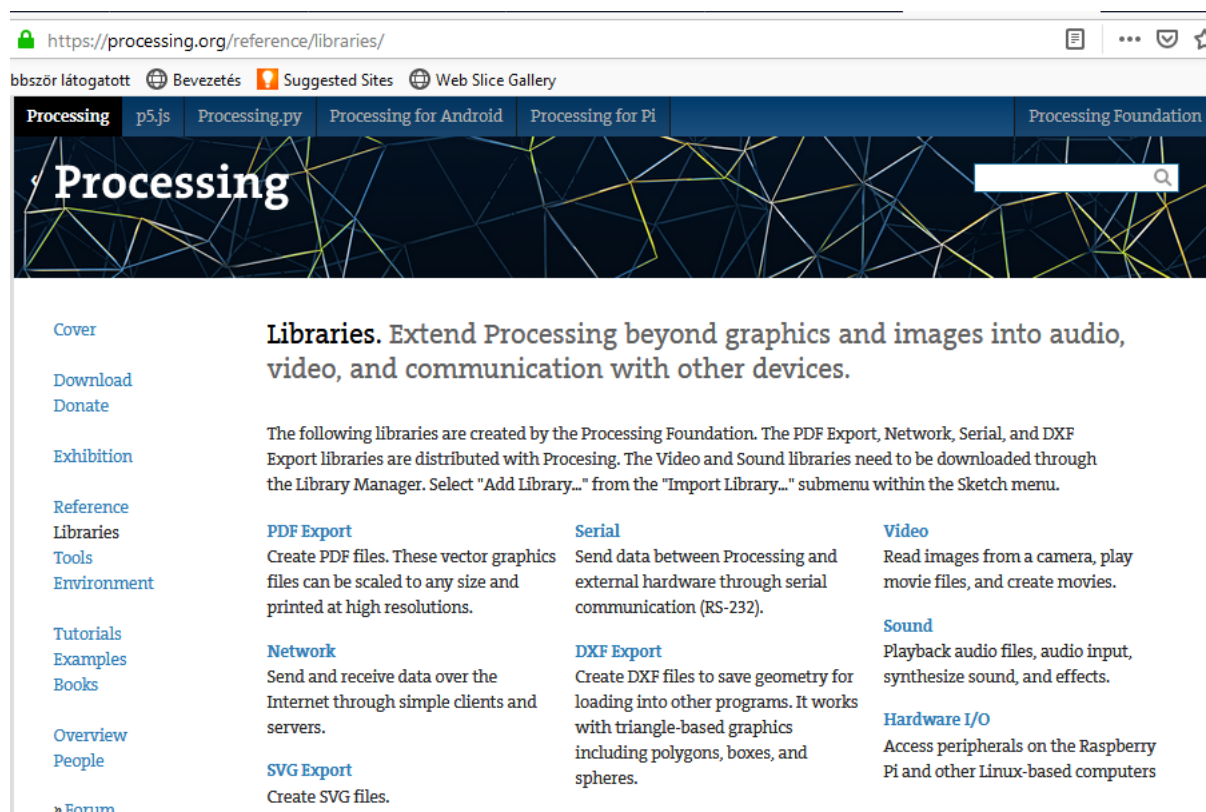
sorokban az aktuális színnel kirajzoljuk a színjelző négyzetet. Ha a setup()-ban helyeztük volna el a felső téglalapok kirajzoltatását, akkor gondoskodnunk kellett volna, hogy felülrajzolásuk esetén újra rajzoltassuk őket, vagy ki kellett volna védeni a belerajzolást. Így egyszerűbb, s rövidebb a kód. A mousePressed() eseményfüggvény kezeli az egér akcióit. A pozícióból derül ki, hogy színválasztás, vagy rajzolás történik:

```
if (mouseX <= 450 && mouseY < 100)
c = get(mouseX, mouseY);
```

A fenti kód második sorában használjuk a get() metódust, mely visszaadja a paramétereiben meghatározott helyzetű pont színét. Ha nem az első sorban megfigyelt területen tartózkodik az egér, akkor a szakasz valamely határolópontjának meghatározása történik a kezd változó értékétől függően. Ha végpontról van szó, akkor a kirajzolás is megtörténik a pontok koordinátáinak és a kezd változó beállításán kívül. A println() függvény csupán a teszteléshez szükséges.

5.3. Szálkezelési lehetőségek, szinkronizálás

Említésre került korábban, hogy a Processing együtt tud működni a Java-val, használhatja az összes osztályát. Ezen felül nagyon hasznosak a speciálisan Processing nyelvhez írt egyéb osztályok, melyet a weblapon baloldalon, a sokat hivatkozott References alatti Libraries menüpontból érhetünk el. Ahogy a 39. ábrán látható, nagyon sok kiterjesztése létezik a nyelvnek.

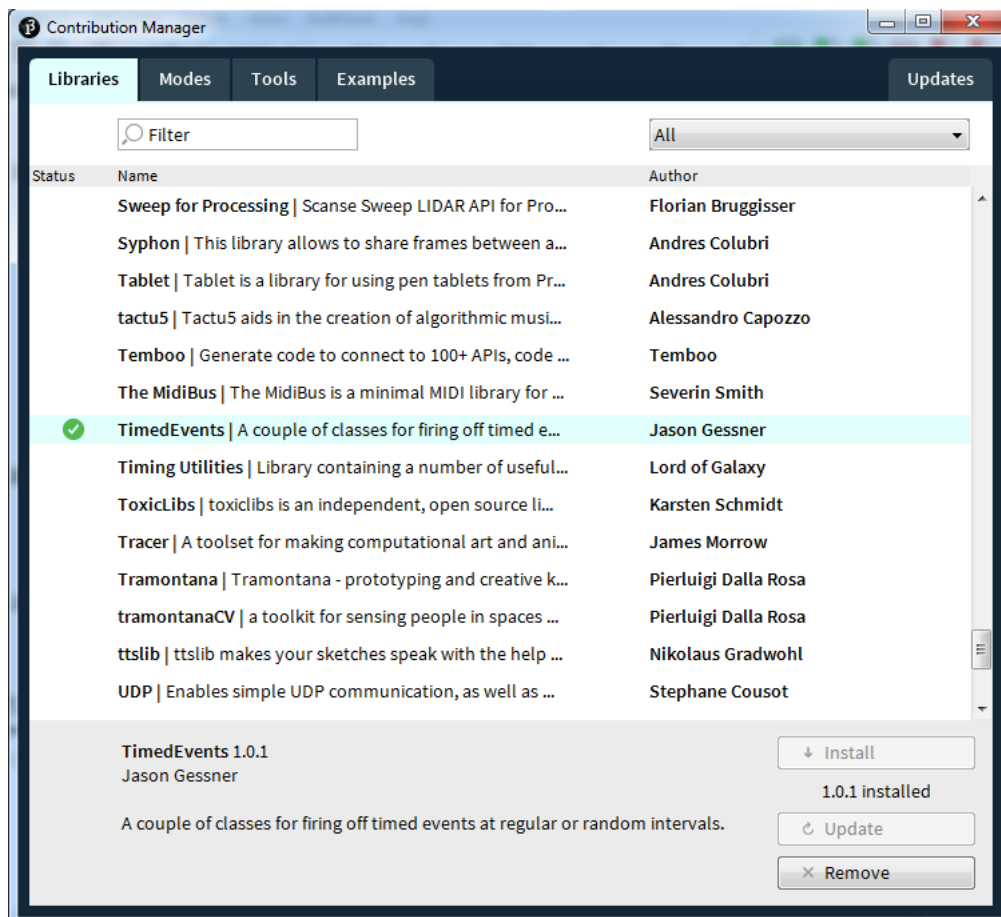


39. ábra. Kiterjesztési lehetőségek

Az adott osztályokat viszont telepíteni kell, nem találhatók meg az "alap" nyelvben. Telepítésüknél legyünk figyelemmel arra, hogy az adott verzióval kompatibilis legyen, különben nem fog működni. Általában kis késéssel követik a nyelv újabb verzióit, de előfordulhat, hogy leáll egy-egy projekt fejlesztése, mivel néhány esetben egyetlen lelkes programozó munkájáról van szó. Azok az osztályok, amelyeket valamely alapítvány, vagy több embert tömörítő társaság kezel, többnyire naprakészek.

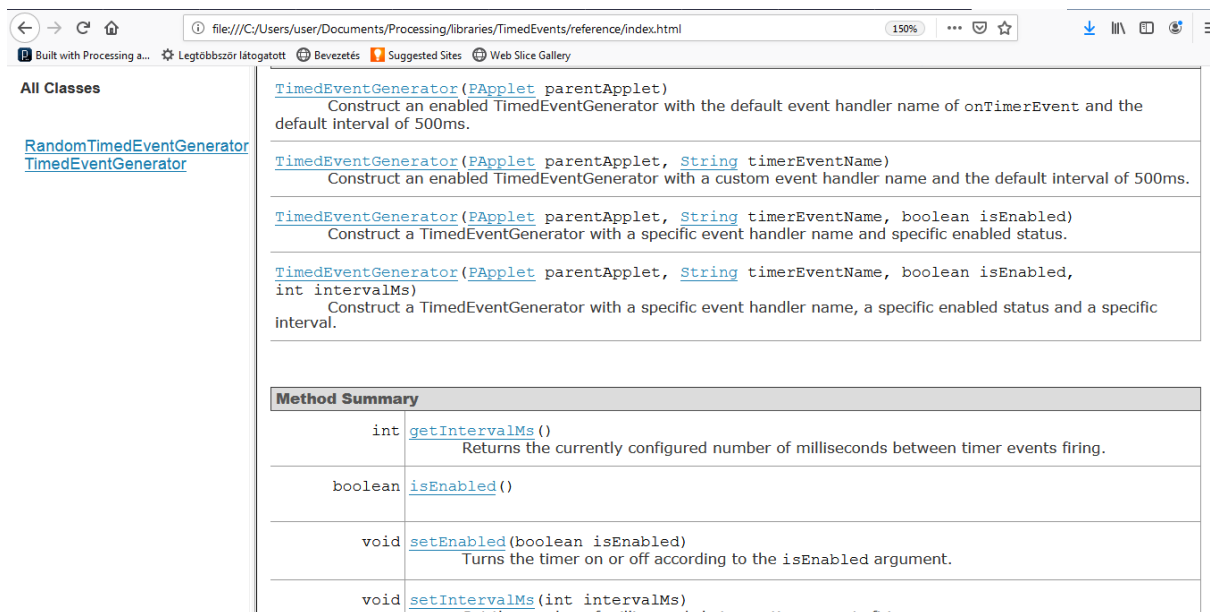
Telepítésük a Processing IDE Tools→ Add Tool... menüpontjából a legegyszerűbb, ekkor megjelenik a 40. ábrán látható Contribution Manager ablak, ahol kiválasztható a szükséges kiterjesztés. Amennyiben nem járunk sikerrel, próbálkozhatunk kézi telepítéssel is, ehhez a <https://github.com/processing/processing/wiki/How-to-Install-a-Contributed-Library> oldalon találunk segítséget.

A szálkezeléshez a TimedEvents bővítést fogjuk használni, ami Jason Gessner munkája. A bővítmény megtalálható Libraries oldalon, valamint a Contribution Manager-ben is. Két osztályt fog tartalmazni, a referenciájukat telepítés után megtaláljuk a bővítmény references mappájában, ahol az index.html állományból kiindulva érdemes megkezdni az ismerkedést. Operációs rendszertől függ a pontos helye, de rákérsva a TimedEvents mappára, könnyen megtalálhatjuk. Megtekintés után látható, hogy összesen két osztályt tartalmaz, ebből a következő feladatban a



40. ábra. A telepítési ablak

TimedEventGenerator-t fogjuk csak használni. A bővítmény referencia állománya a 41. ábrán tekinthető meg. Az alábbi program ismertetésével a bővítményekre irányuló figyelemfelhívás



41. ábra. A TimedEvents referencia lapok egy részlete

a célunk, érdemes körbenézni, hogy kész, avagy hasznos megoldásokat találjunk.

A program a fentebb említett osztály segítségével két időzített-esemény objektumot hoz létre, melyek egy-egy függvényhez kapcsolódnak. Mindkét függvény számokat ír ki a grafikus felületre: Az `iro_1()` 1000-től kezdve ír, minden kiíráskor eggyel növelve, az `iro_2()` pedig az aktuális másodpercet jeleníti meg meghívása esetén. Azt akartuk elérni, hogy az `iro_1()` meghívásának gyakoriságát a felhasználó tudja kontrollálni, míg az `iro_2()` csak két másodpercenként kerüljön lefuttatásra. A feladatot a `TimedEvent` bővítmény `TimedEventGenerator` osztálya segítségével oldottuk meg. A forráskód alább, míg a végrehajtás eredménye a 42. ábrán látható.

```
import org.multiply.processing.TimedEventGenerator;

private TimedEventGenerator ir_1, ir_2;
private int ido=1000;
void setup() {
    size(500, 200);
    textSize(50);
    background(0);
    ir_1=new TimedEventGenerator(this, "iro_1");
    ir_1.setIntervalMs(100);
    ir_2=new TimedEventGenerator(this, "iro_2");
    ir_2.setIntervalMs(2000);
}

void draw() {
}

void iro_1() {
    fill(0);
    rect(0, 0, 300, 200);
    fill(255);
    text(ido++, 100, 100);
}

void iro_2(){
    fill(0);
    rect(300, 0, 500, 200);
    fill(255);
    text(second(), 300, 100);
}

void keyPressed() {
    int x=ir_1.getIntervalMs();
    if (key=='a') ir_1.setIntervalMs(x+10);
    else if (x>10) ir_1.setIntervalMs(x-10);
    println(x);
}
```

Az

```
import org.multiply.processing.TimedEventGenerator;
```

sorral tudatjuk, hogy külső osztályt fogunk használni, ebből két objektumot hozunk létre, melyekhez hozzákötjük a megfelelő meghívandó függvényeket. A bekötött függvények működését nem részletezzük, már megismert dolgokat tartalmaznak:

```
ir_1=new TimedEventGenerator(this, "iro_1");  
ir_2=new TimedEventGenerator(this, "iro_2");
```

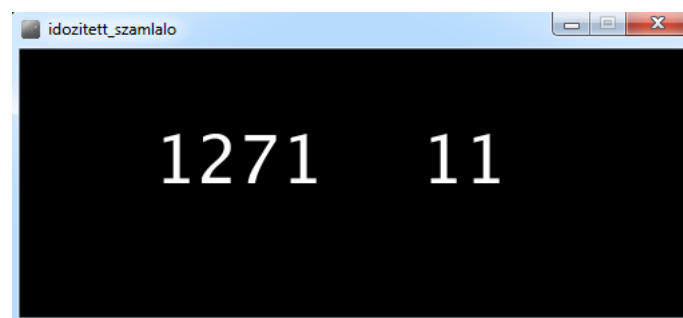
A meghívásuk gyakoriságát az

```
ir_1.setIntervalMs(100);  
ir_2.setIntervalMs(2000);
```

sorokkal álltjuk be. A billentyűzet lenyomásakor az

```
int x=ir_1.getIntervalMs();  
if (key=='a')  
    ir_1.setIntervalMs(x+10);  
else  
    if (x>10)  
        ir_1.setIntervalMs(x-10);  
println(x);
```

sorok futnak le. A `getIntervalMs()` visszaadja a hívás gyakoriságát ezredmásodpercben, s látható, hogy az *a* lenyomására nőni fog a meghívások közötti időtartam egy századmásodperccel, más karakter esetén ugyanennyivel csökken. A `println()` itt is csak tesztelési célokat szolgál. Szinkronizálni ezeket a folyamatokat pl. globális változón keresztül lehetne, ami nagyon nem elegáns. A megoldást az öröklés szolgáltatja, ekkor ki lehet bővíteni a metódusok formálisparaméter listáit a szinkronizálást biztosító referencia típusú paraméterrel. Primitív típusú nem megfelelő, mivel értékszerinti paraméterátadás van a Processing-ben/Java-ban. A program a Processing 2.2.1 + TimedEvents 1.0.1 verzióval volt tesztelve.



42. ábra. Időzített számláló

6. Összefoglalás

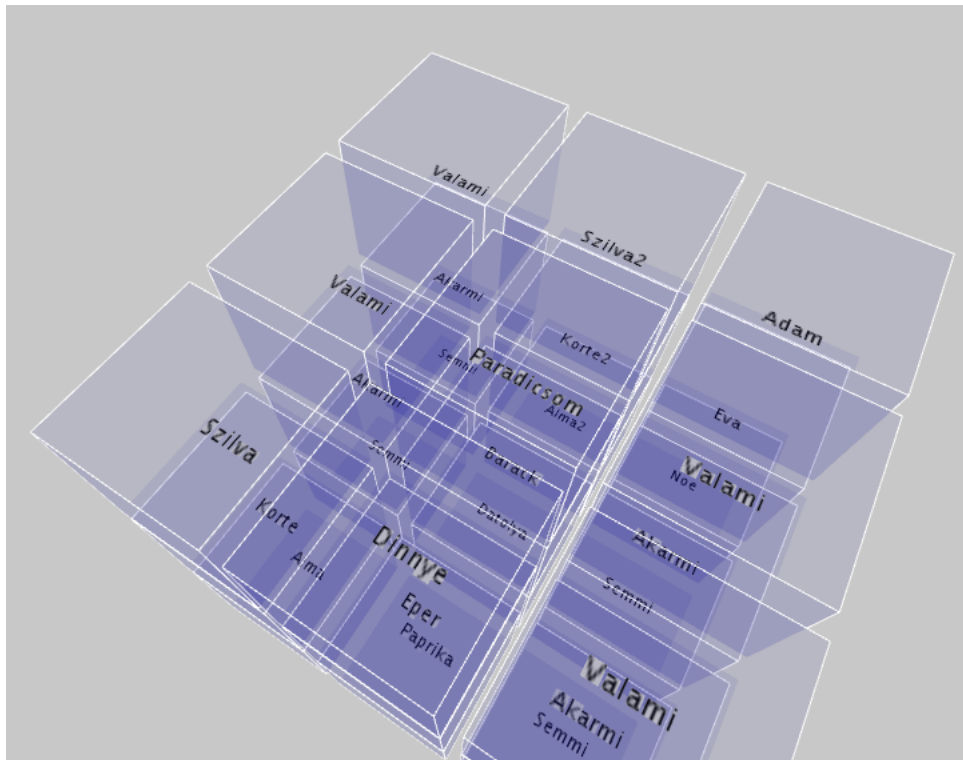
A jegyzet célja kettős, egyrészt a Processing nyelvet hivatott bemutatni, mint a Java-ban történő grafikus felületen folyó programozási lehetőségét, másrészt az időhöz kötött feladatok megoldását. Csak bevezetés szintjén tudtuk ismertetni a nyelvet a terjedelem miatt, a referencia további lehetőségeket rejt, ahogy a Java nyelv, és az előző fejezetben ismertetett bővítmények. Demonstrálandó a benne rejlő további képességeket, a következő programkódot magyarázat nélkül közöljük. Egy átlátszó kockákból álló kockát tudunk megjeleníteni általa, melyben a forrás elején megadott tömbben lévő szövegek jelennek meg. A kocka egérrel forgatható és elmozdulás közben újraszámolódnak valós időben a takarások perspektivikusan. A program eredménye a 43. ábrán látható.

```
int a=80, g=10;
PShape[] t=new PShape[3];
color[] c={color(255, 0, 0), color(0, 255, 0), color(0, 0, 255)};
String sz[][][]={
  {"Semmi", "Akarmi", "Valami"}, {"Semmi", "Akarmi", "Valami"}, {"Alma", "Korte", "Szilva"}},
  {"Alma2", "Korte2", "Szilva2"}, {"Datolya", "Barack", "Paradicsom"}, {"Paprika", "Eper", "Dinnye"}},
  {"Noe", "Eva", "Adam"}, {"Semmi", "Akarmi", "Valami"}, {"Semmi", "Akarmi", "Valami"}}
};
PShape [][] r=new PShape[3][3];
void setup() {
  size(800, 600, P3D);
  background(color(0, 0, 200));
  textSize(a/8);
  textAlign(CENTER);
  for (int k=0; k<3; k++) {
    t[k]=createShape(GROUP);
    for (int i=0; i<3; i++)
      for (int j=0; j<3; j++) {
        r[i][j]=createShape(BOX, a);
        r[i][j].translate(i*(a+g), j*(a+g), k*(a+g));
        r[i][j].setFill(color(0, 0, 150, 20));
        r[i][j].setStroke(color(255, 255, 255));
        t[k].addChild((r[i][j]));
      }
  }
}
void draw() {
  background(200);
  translate(width/2, height/2, 0);
  rotateX(radians((mouseY/3)));
  rotateZ(radians(mouseX/3));
  for (int k=0; k<3; k++) {
```

```

fill(0);
for (int i=0; i<3; i++)
for (int j=0; j<3; j++)
text(sz[i][j][k], i*(a+g), j*(a+g), k*(a+g));
}
shape(t[0]);
shape(t[1]);
shape(t[2]);
}

```



43. ábra. 3D-s test és szöveg forgatása átfedésekkel

Hivatkozások

- [1] Processing weboldala: [https://processing.org/https://processing.org/](https://processing.org/)
- [2] Juhász Ferencné: Valósídejű programozás, LSI Oktatóközpont, Budapest, 1999
- [3] Angster Erzsébet: Objektumorientált tervezés és programozás Java, 4KÖR Bt, Budapest, 2001.